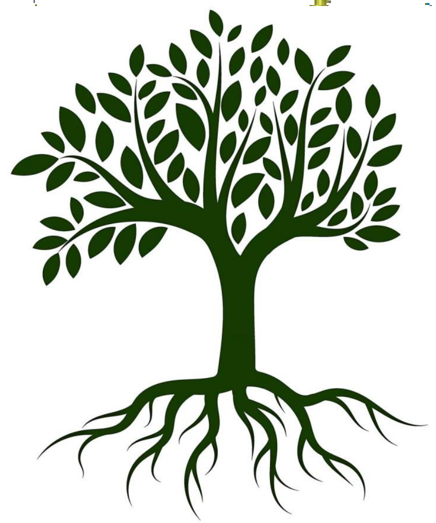
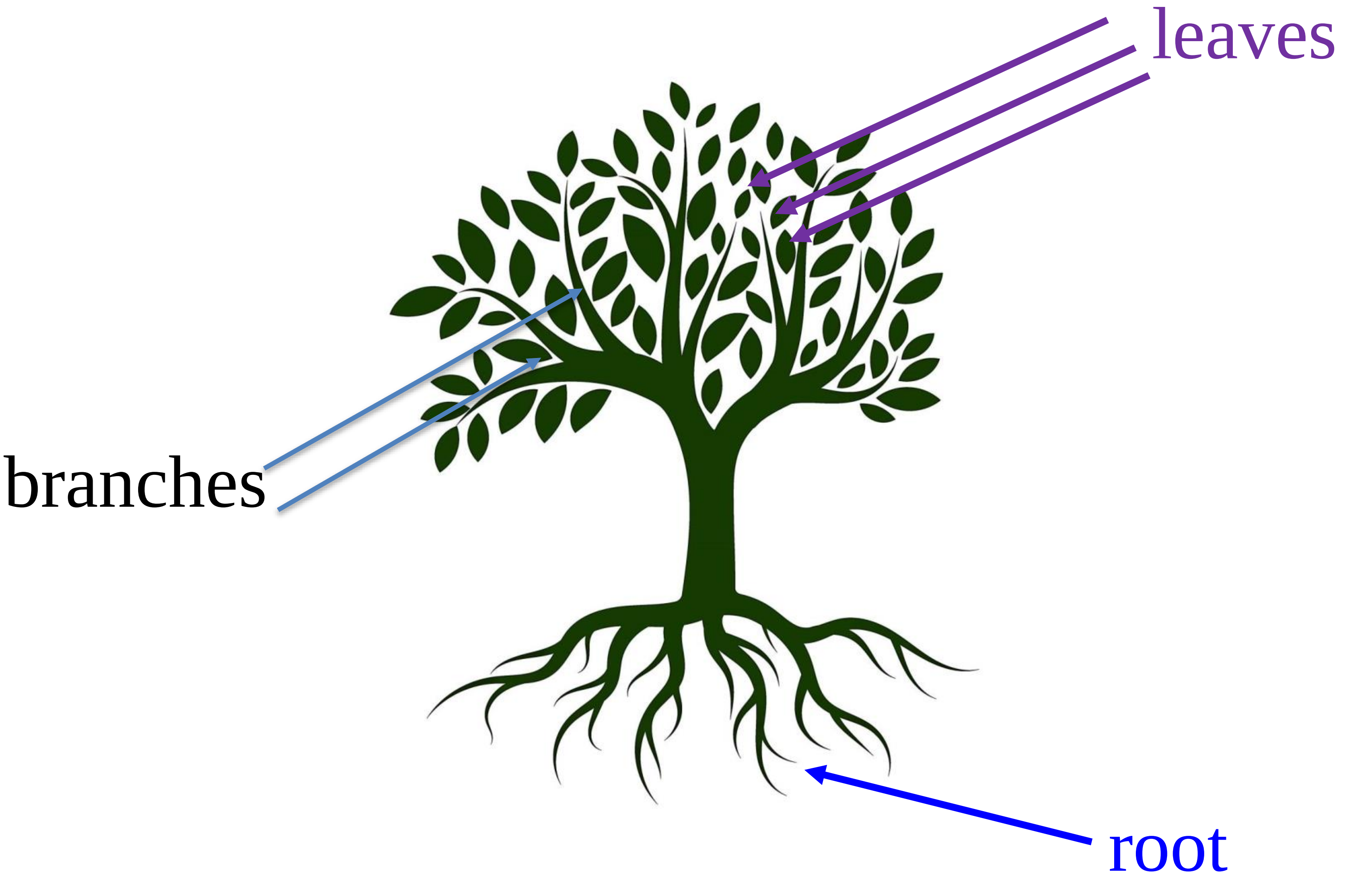
SEHH2239
Data Structures

Lecture 8

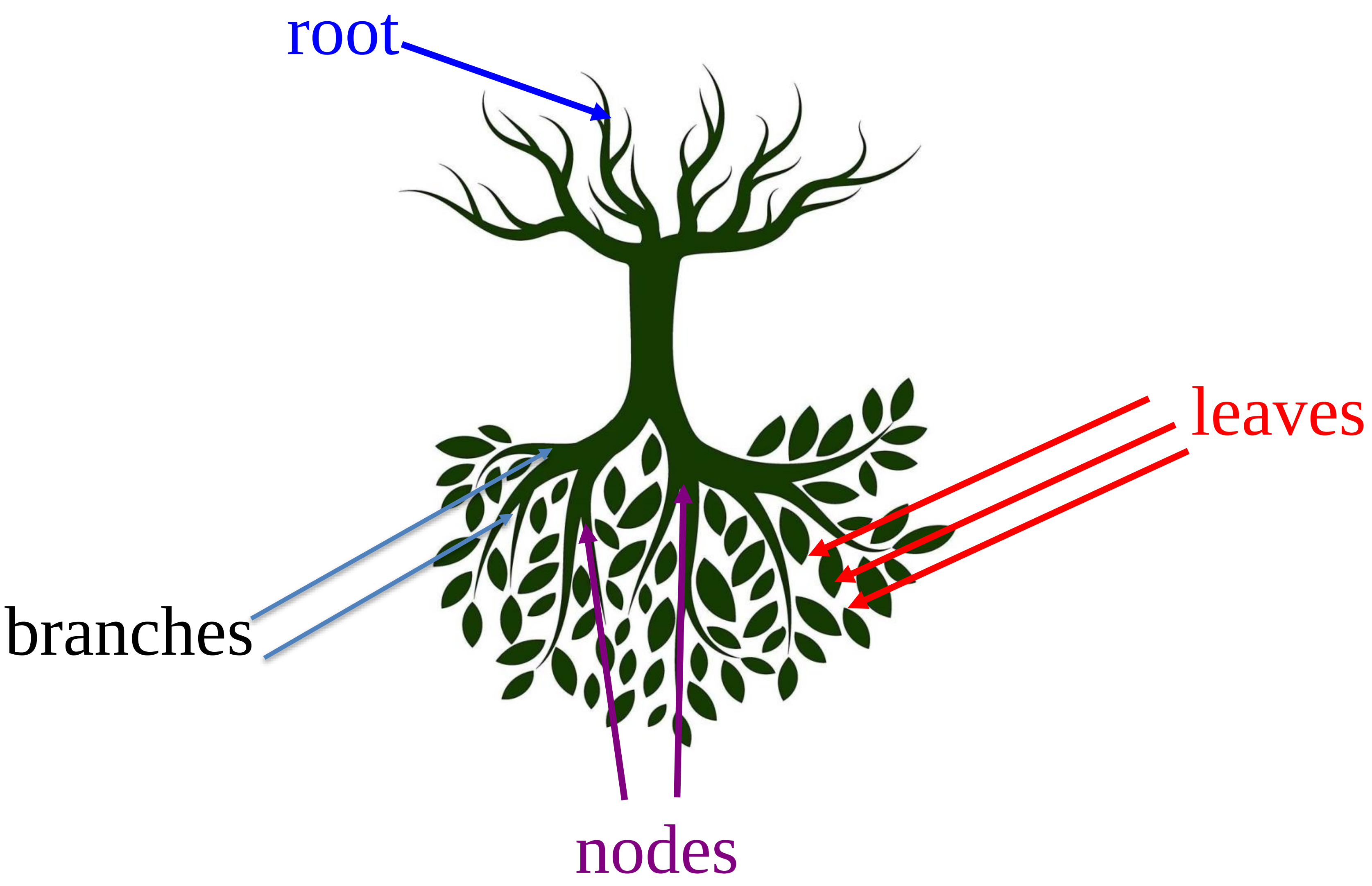
Tree I



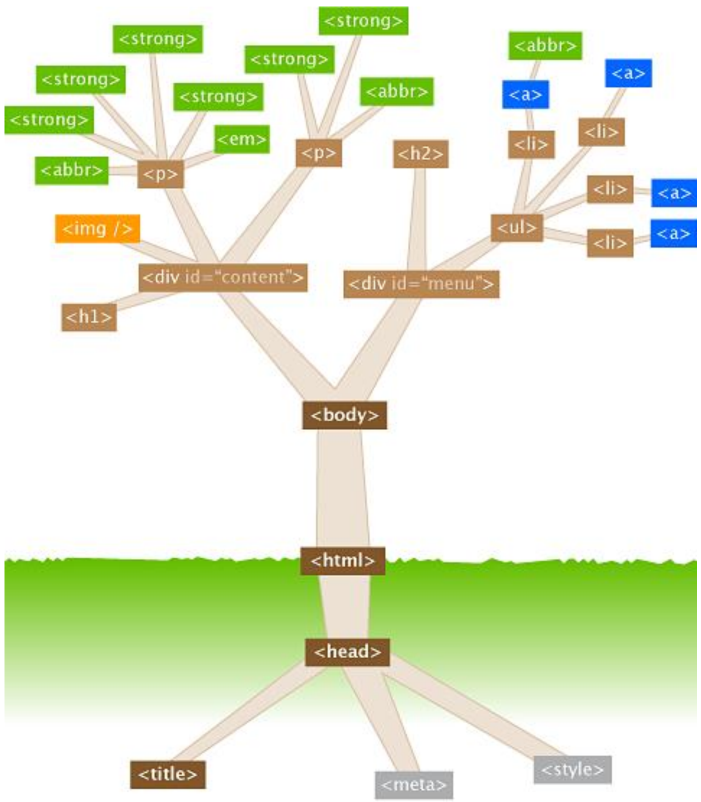
Nature Lover's View Of A Tree



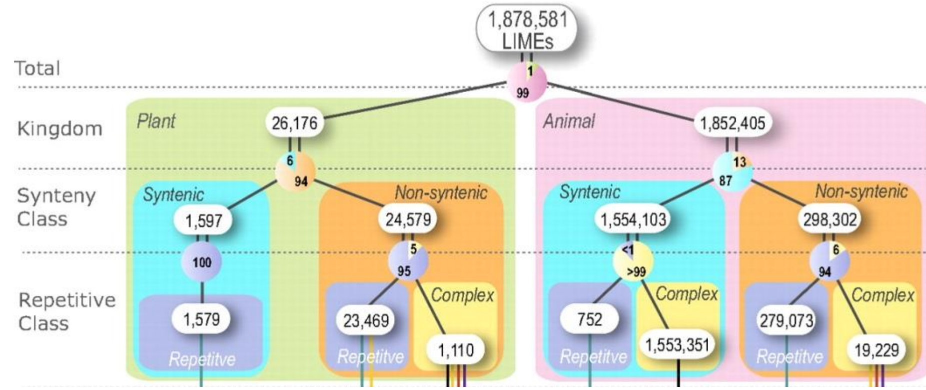
Computer Scientist's View



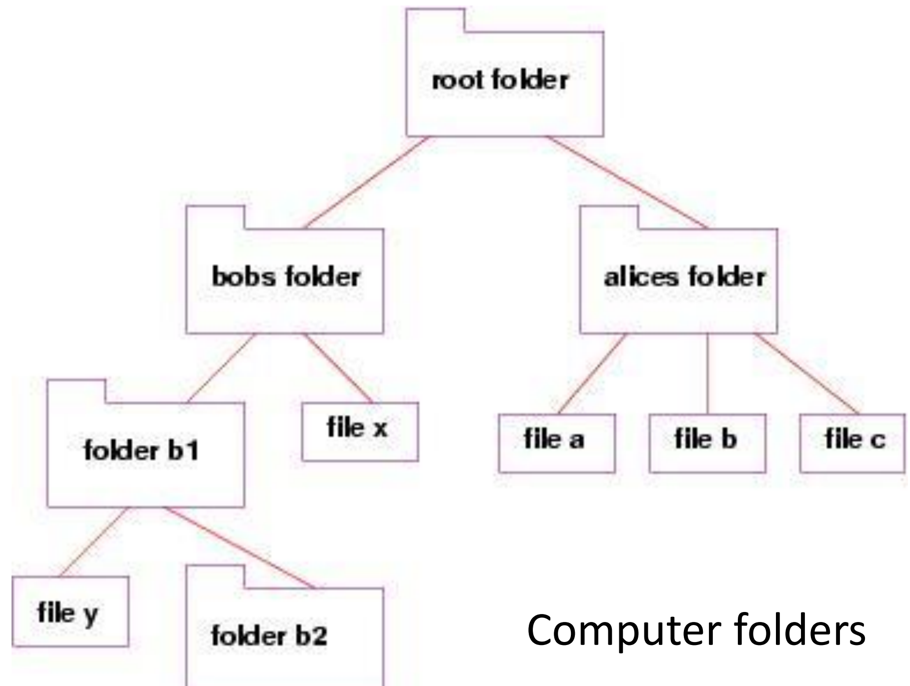
Example of Trees



HTML Document Structure



Organism Classifications

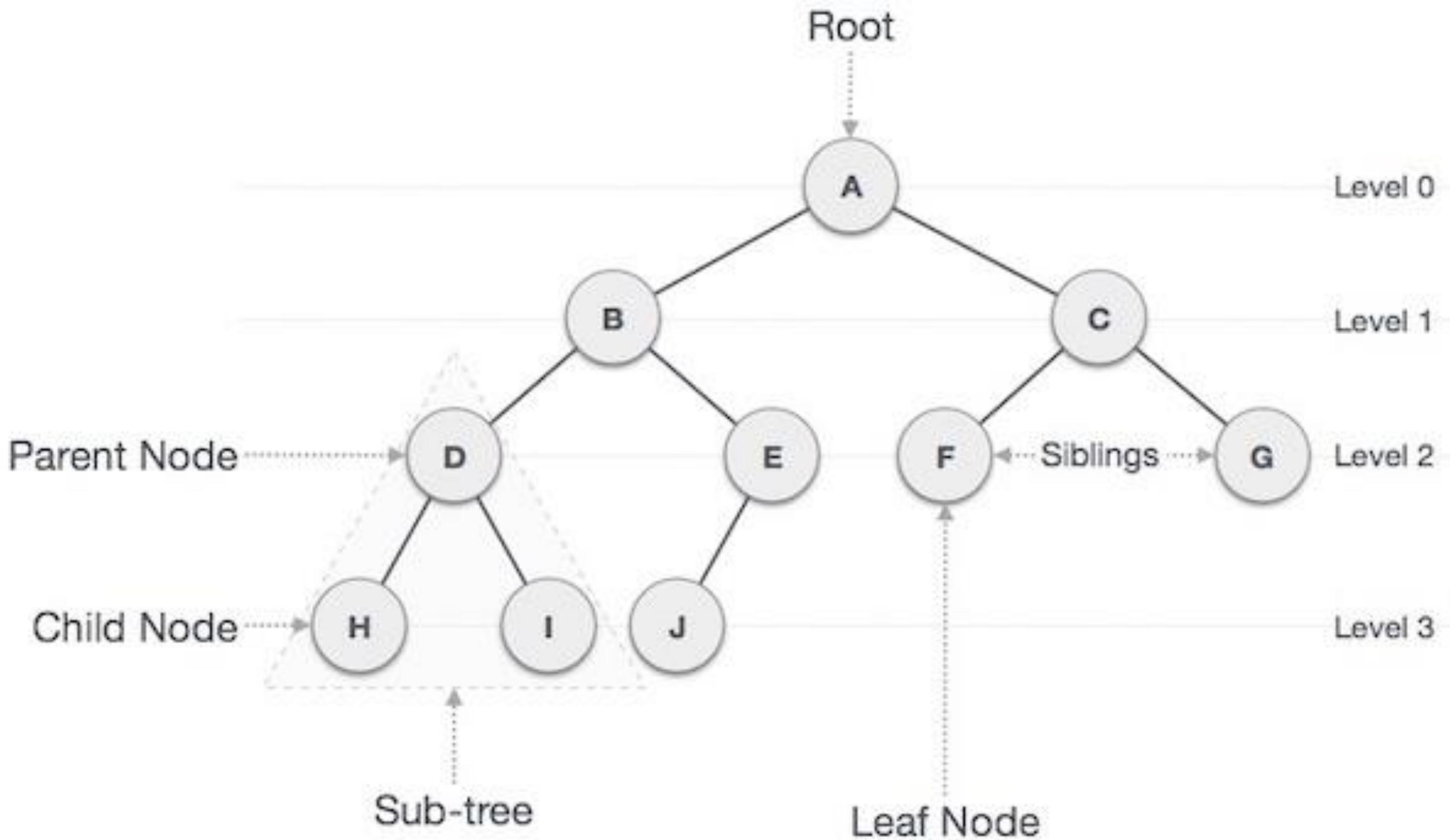


Computer folders

Definition

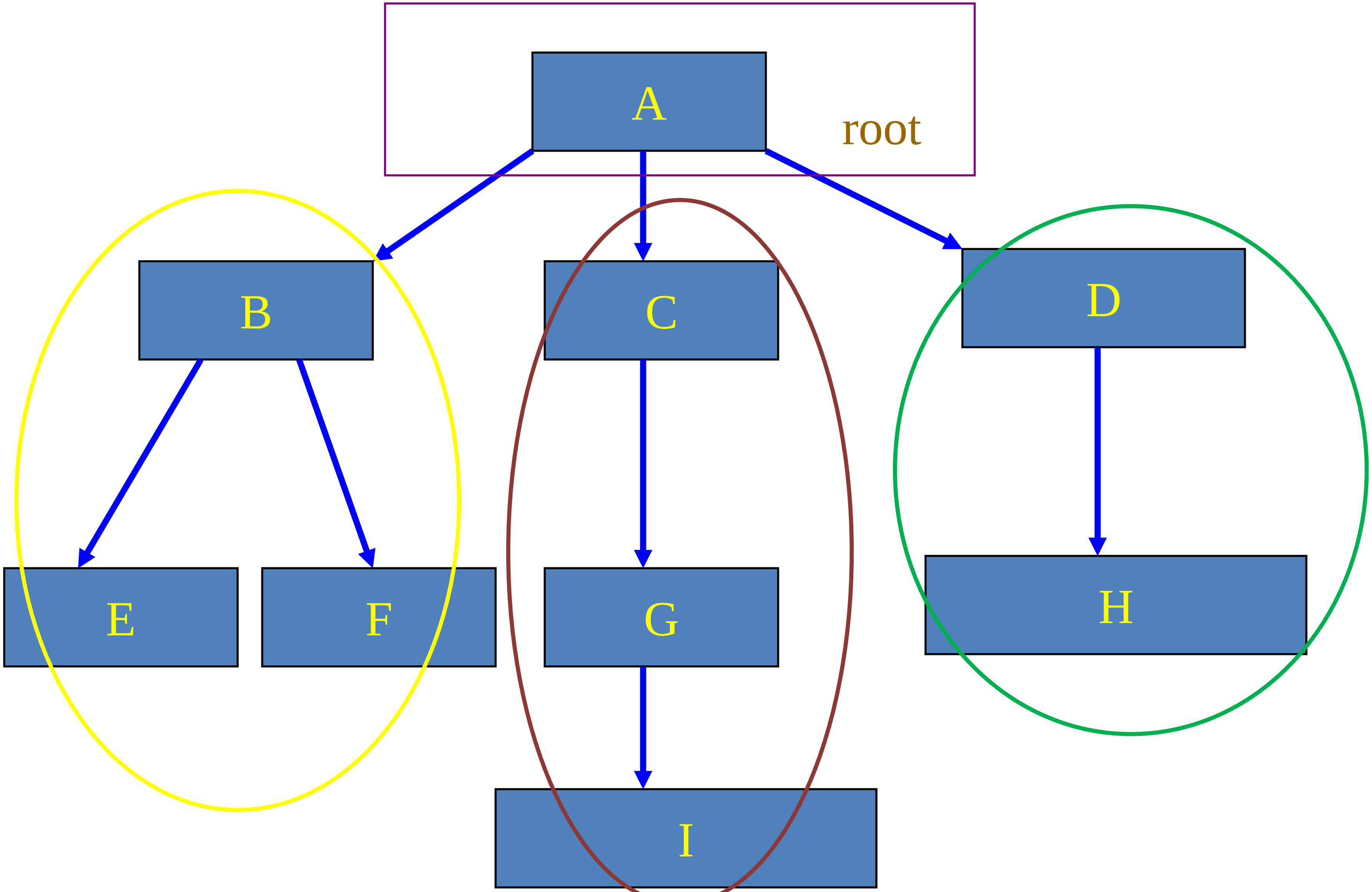
- A tree ***t*** is a finite nonempty set of elements.
- One of these elements is called the **root**.
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of ***t*** .

Tree

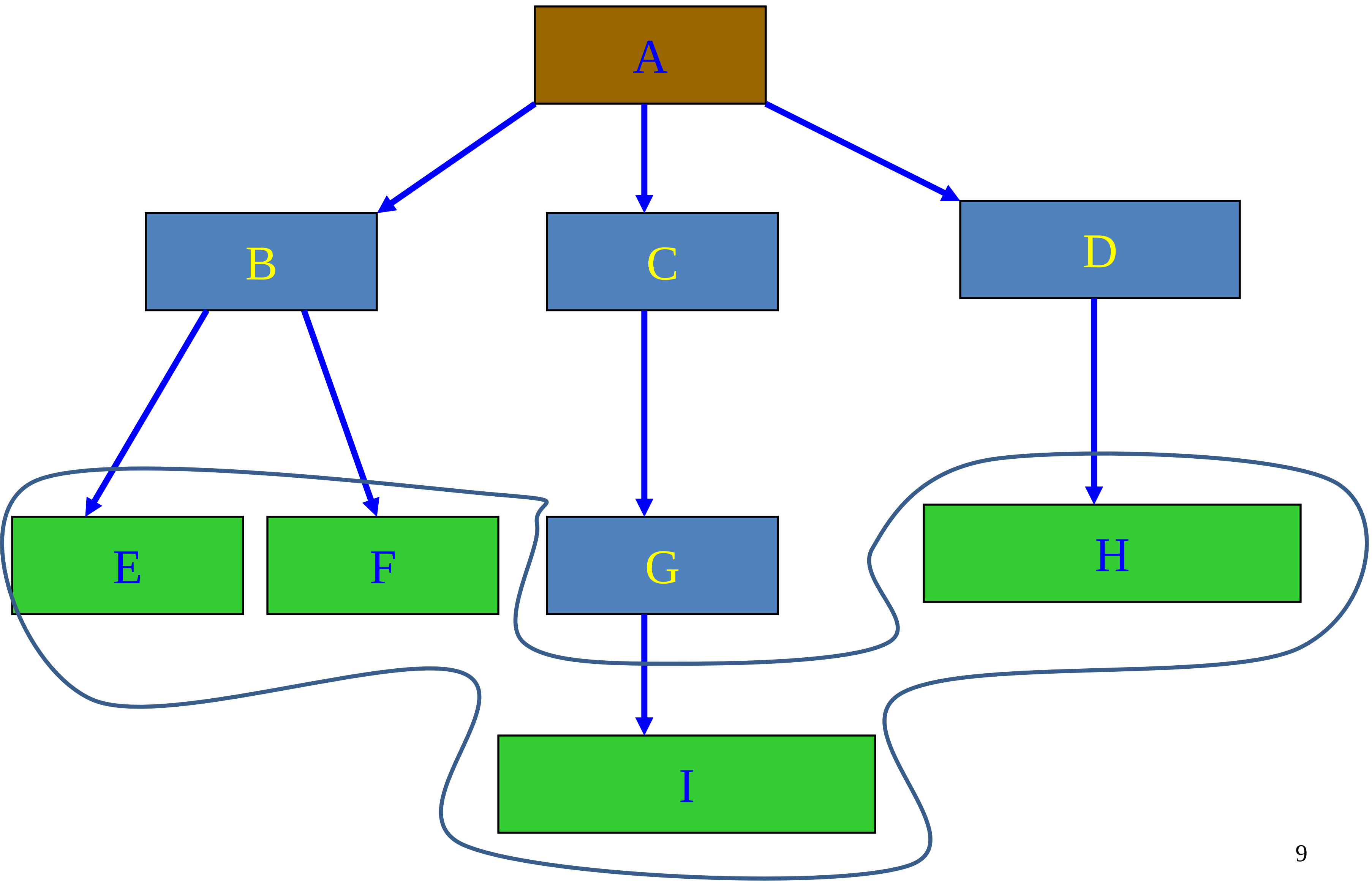


https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

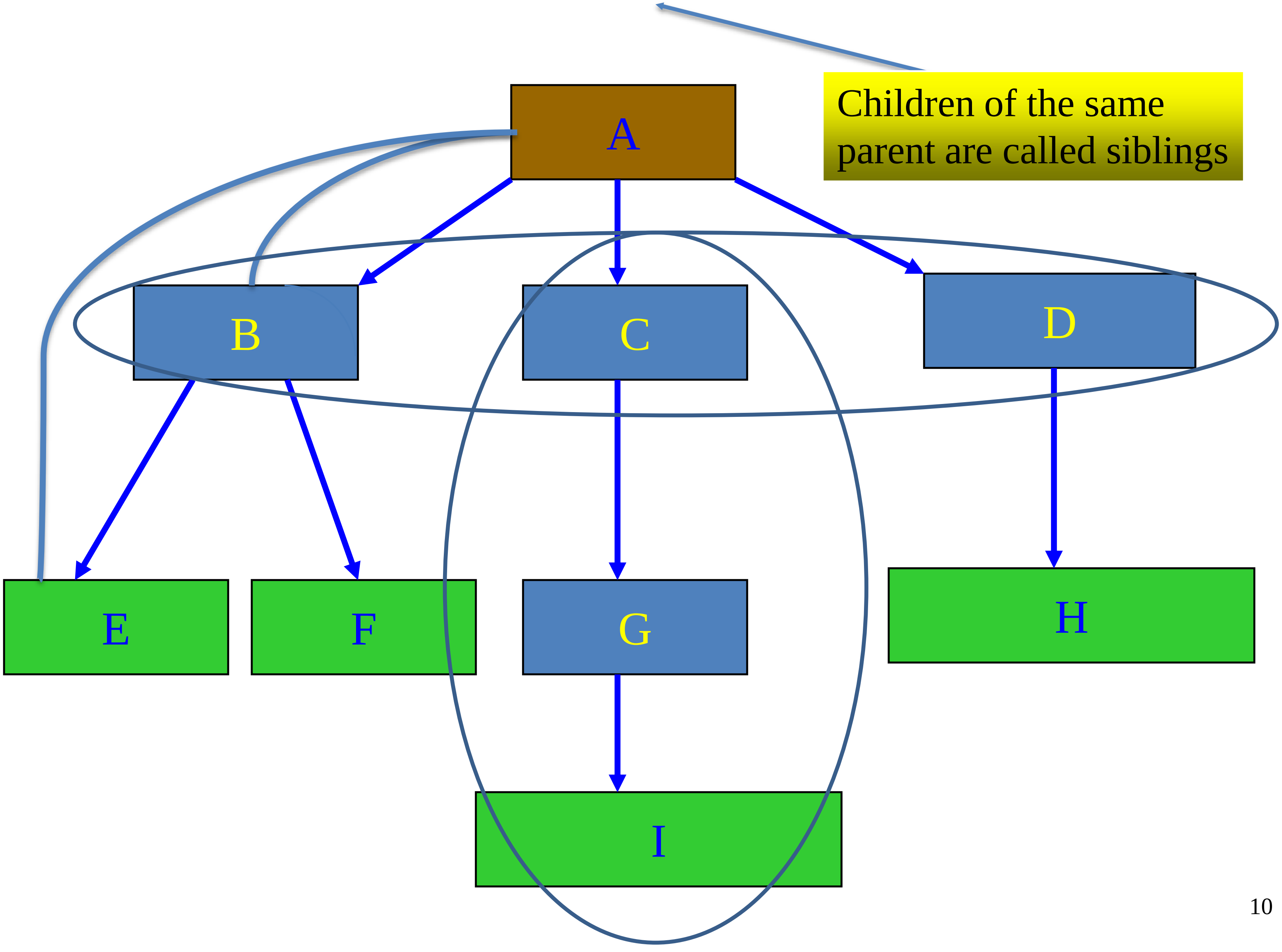
Subtrees



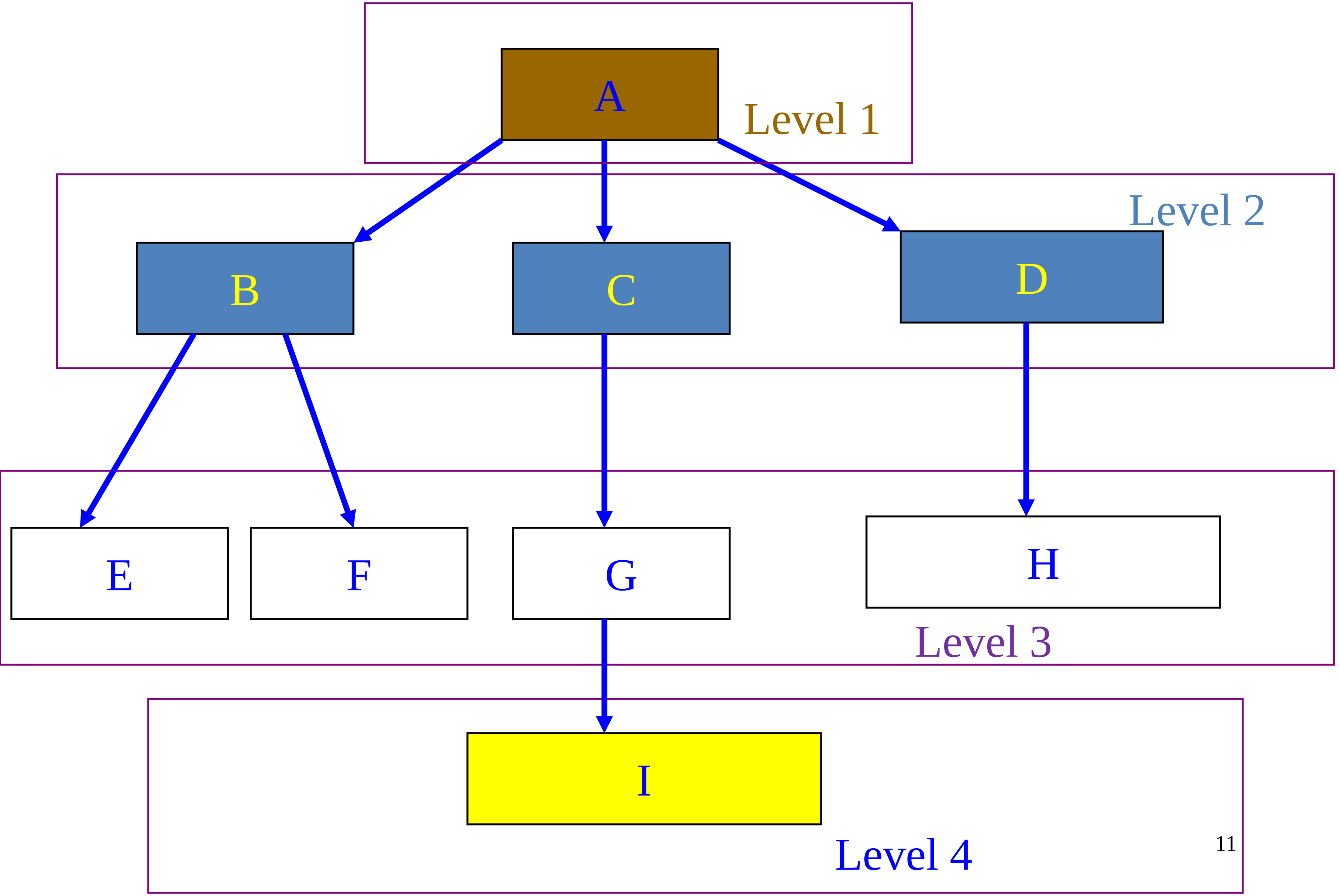
Leaves



Parent, Grandparent, Siblings, Ancestors, Descendants



Levels



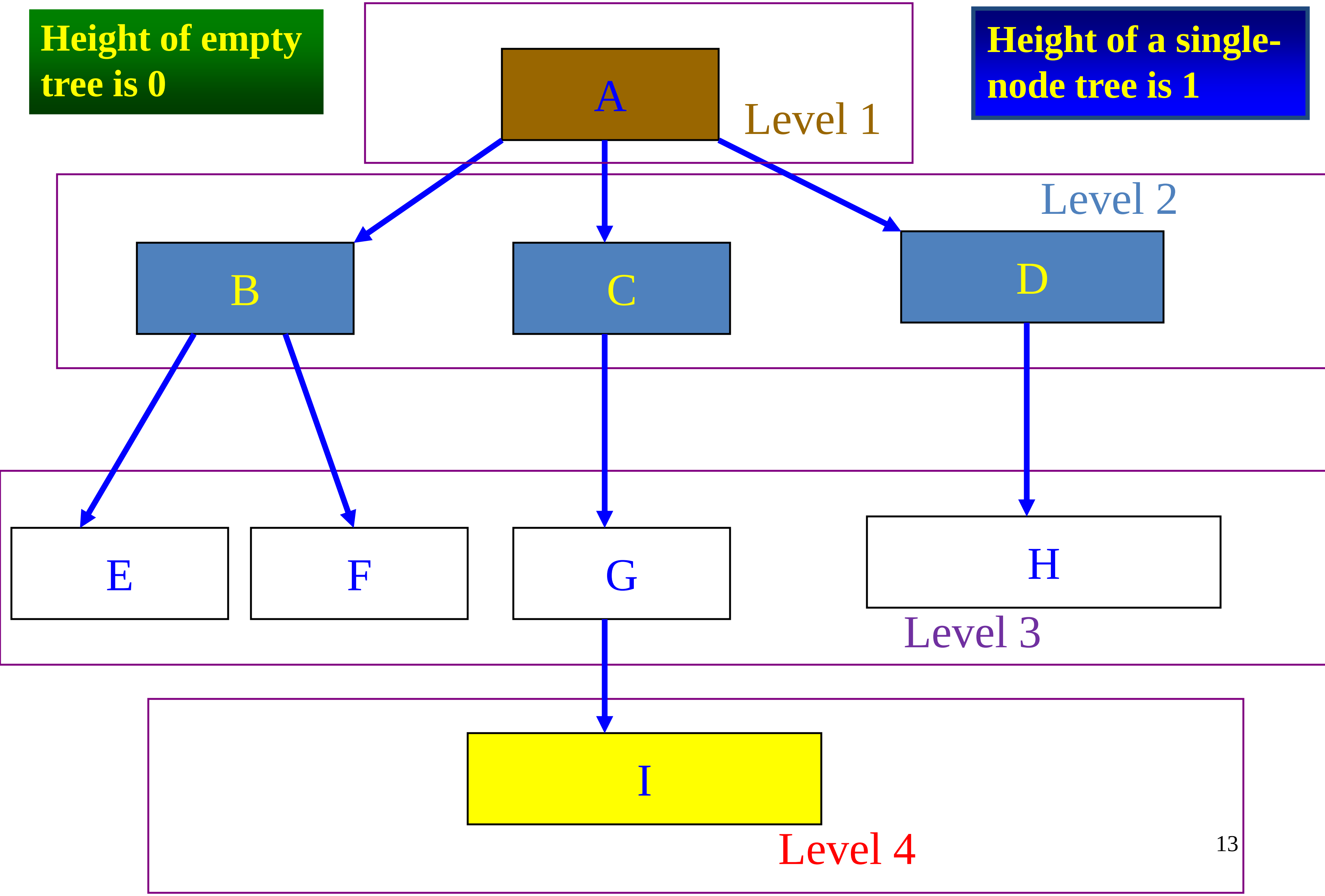
Caution

- Some texts start level numbers at 0 rather than at 1 (e.g. The textbook by Hubbard and Huray)
- We shall number levels as follows:
- Root is at level 1.
- Its children are at level 2.
- The grand children of the root are at level 3.
- And so on.

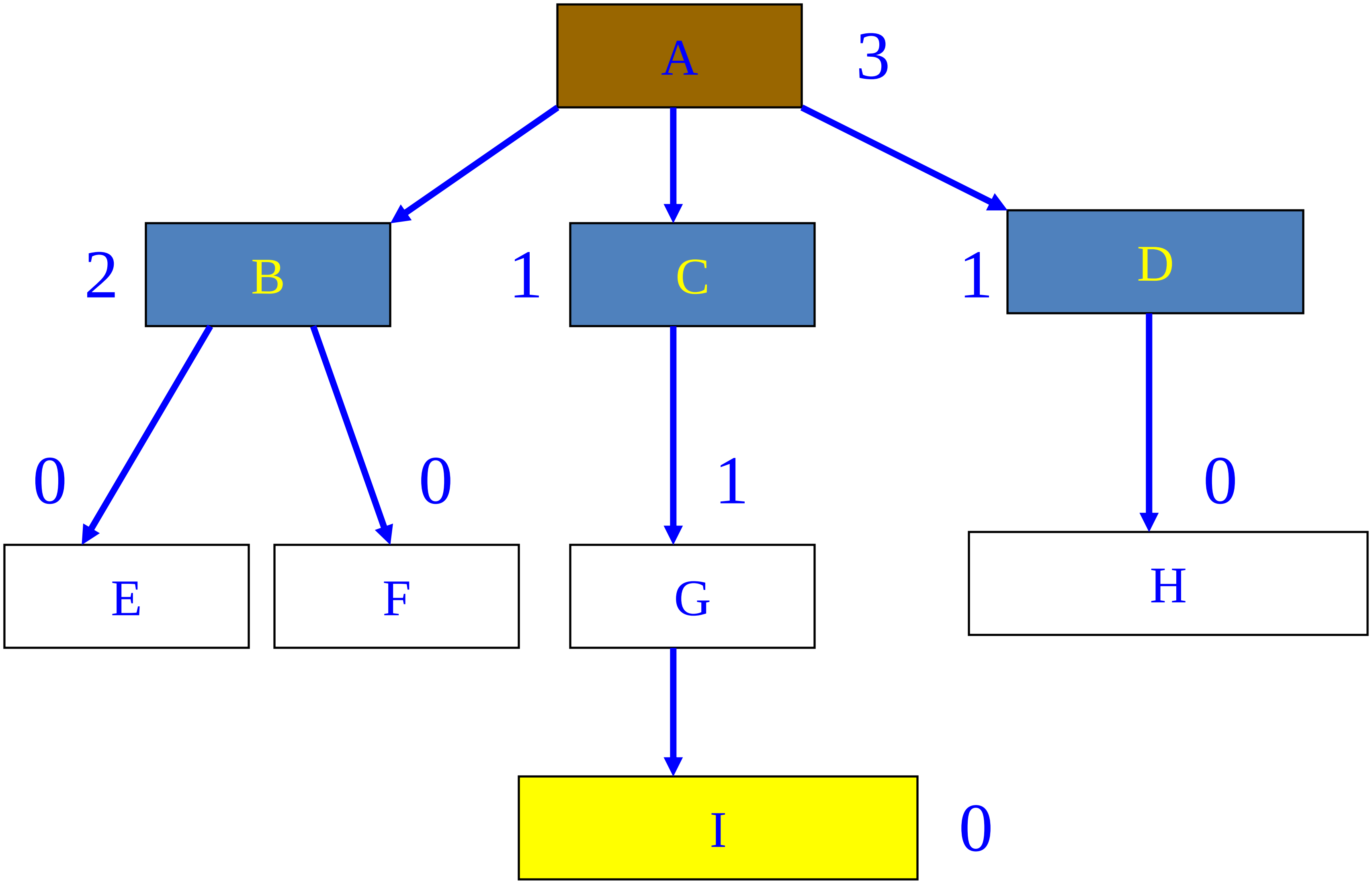
height = depth = highest level number

Height of empty tree is 0

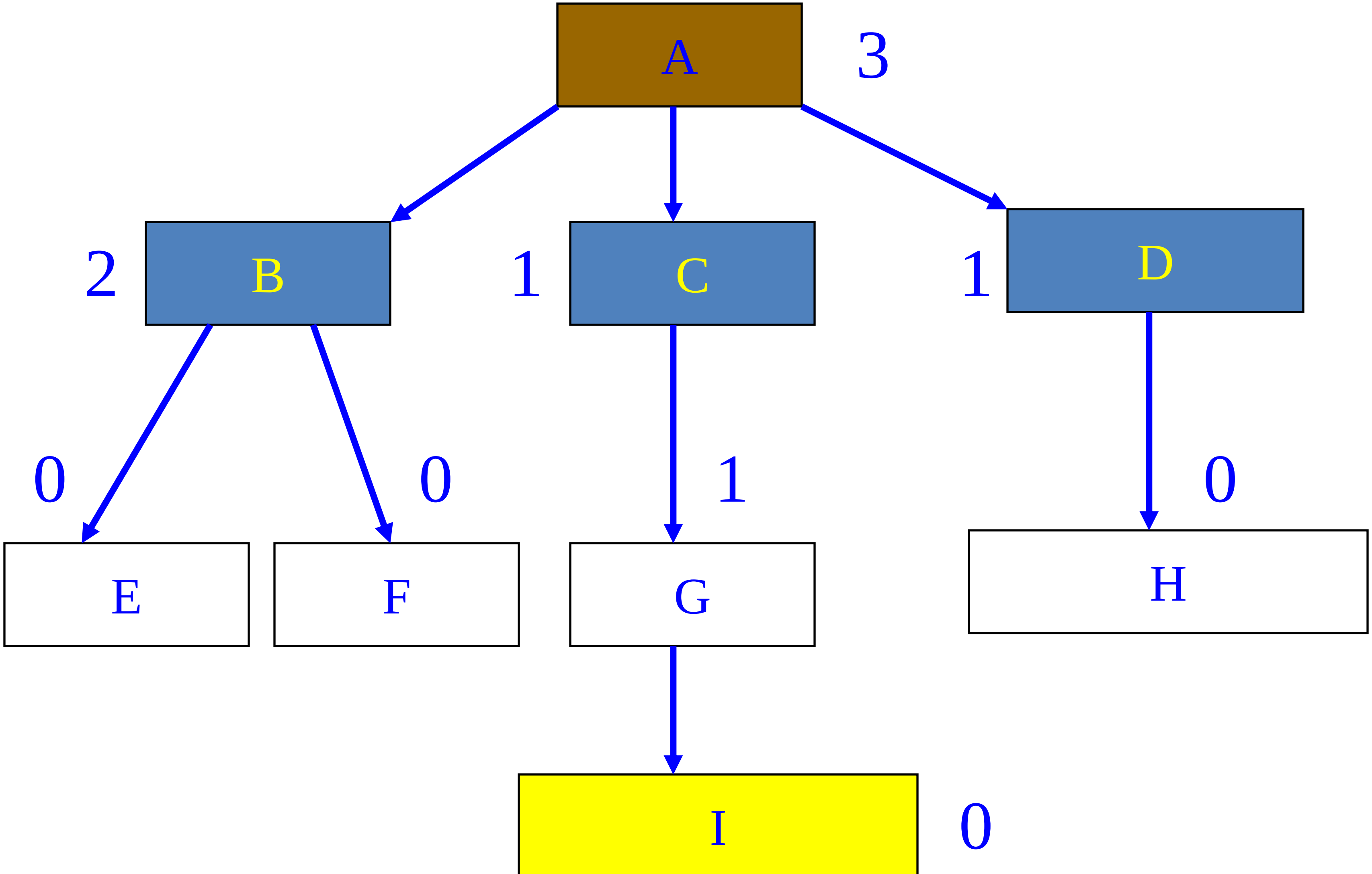
Height of a single-node tree is 1



Node Degree = Number Of Children



Tree Degree = Max Node Degree

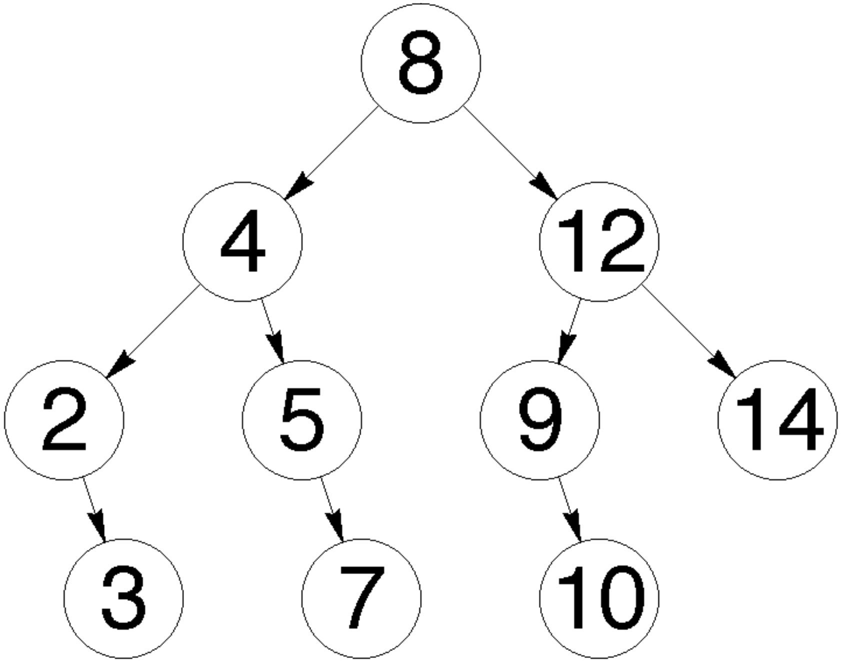


Degree of tree = 3.

Tree - Exercise

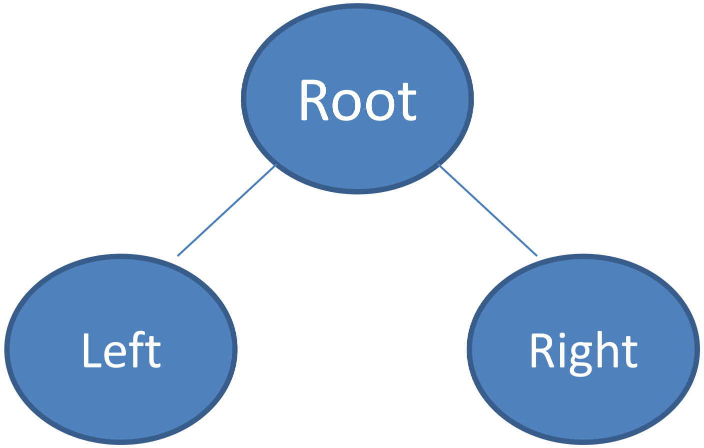
- Find the followings:

1. Root
2. Leave
3. Siblings
4. Height
5. Depth
6. Tree Degree



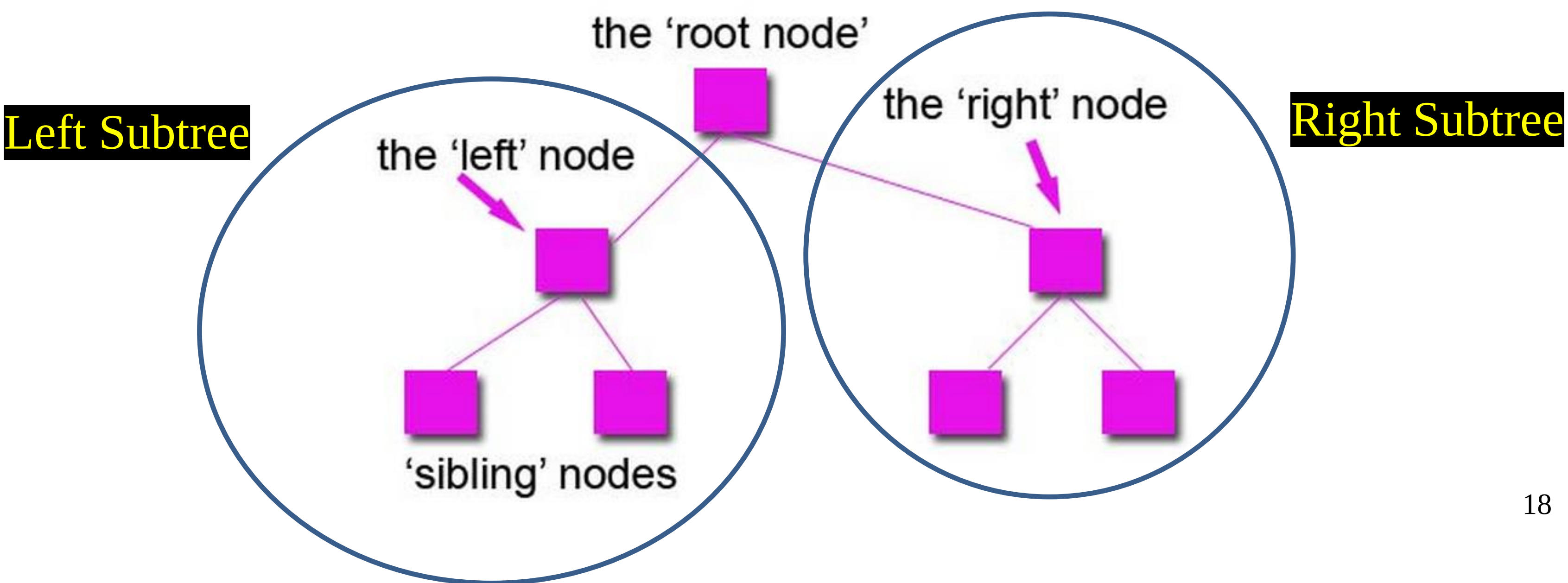
Binary Tree

- Finite (possibly empty) collection of elements.
- A nonempty binary tree has a root element.
- The remaining elements (if any) are partitioned into *two binary trees*.
- These are called the left and right subtrees of the binary tree.



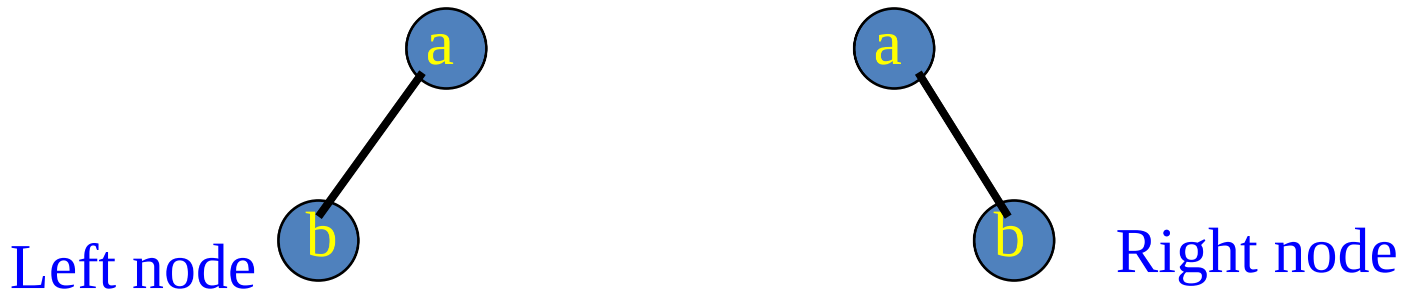
Differences Between A Tree & A Binary Tree

- **Binary tree:**
  - **No node may have a degree more than 2.**
- **Tree:**
  - **No limit** on the degree of a node in a tree.



Differences Between A Tree & A Binary Tree

- The subtrees of a binary tree are **ordered**; those of a tree are not ordered.



- Are different when viewed as binary trees.
- Are the same when viewed as trees.

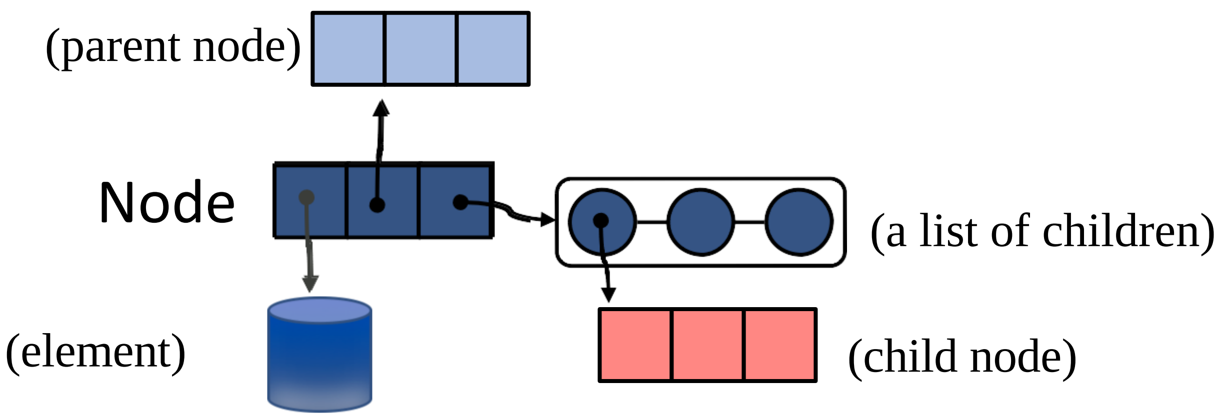
Tree Implementation

Structure for a Tree

A **Node** is represented by an object

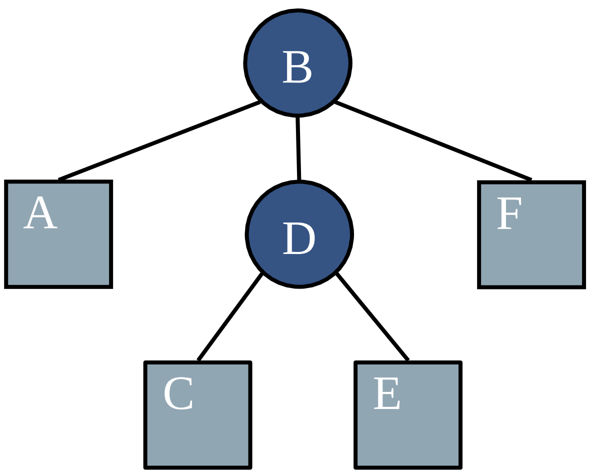
Data variables:

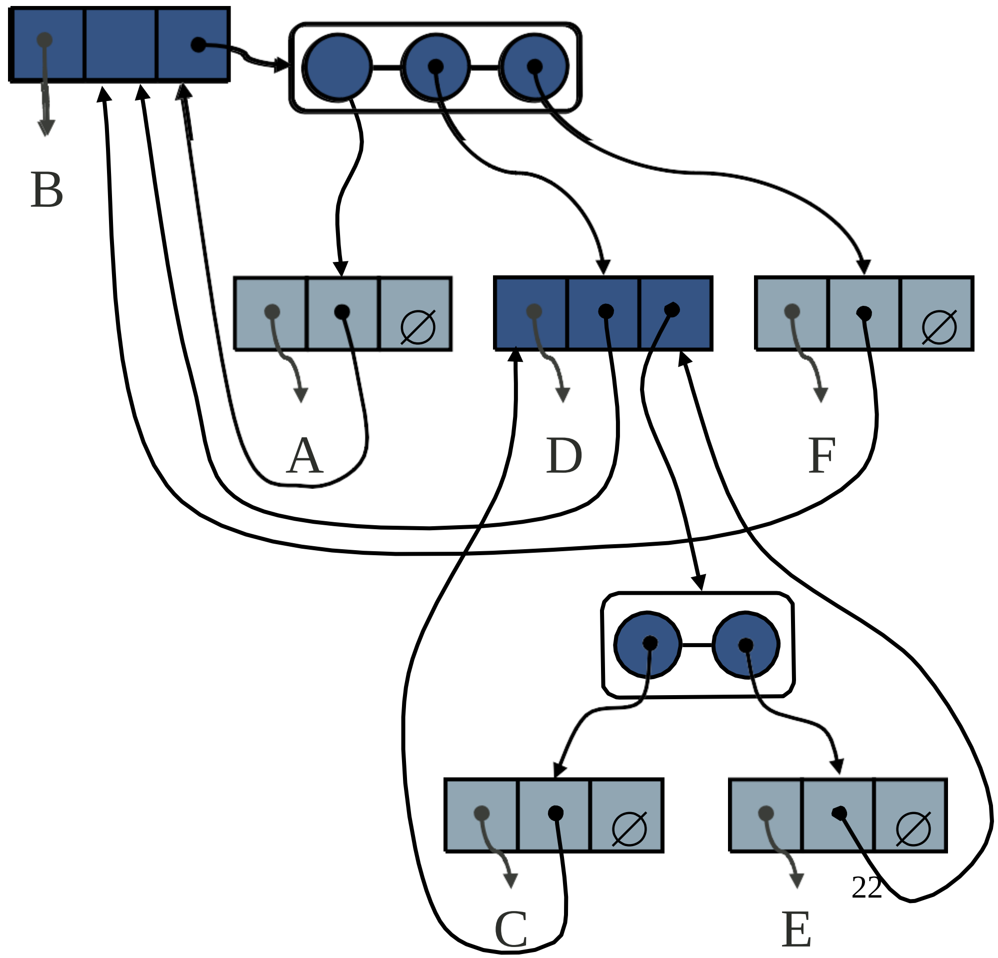
- Element
- A parent node
- A sequence of children nodes



Structure for a Tree

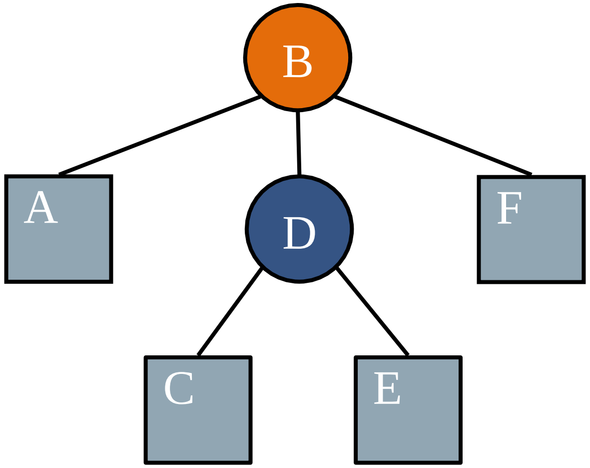
Try to implement the following Tree:



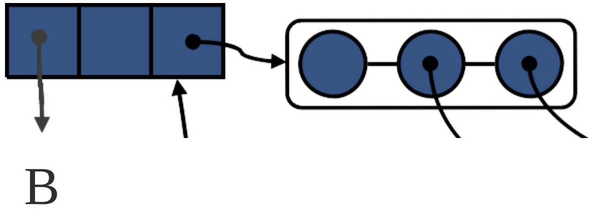


Structure for a Tree
- Create the root

```
class Tree:
    def __init__(self, data):
        self.children = []
        self.data = data
```

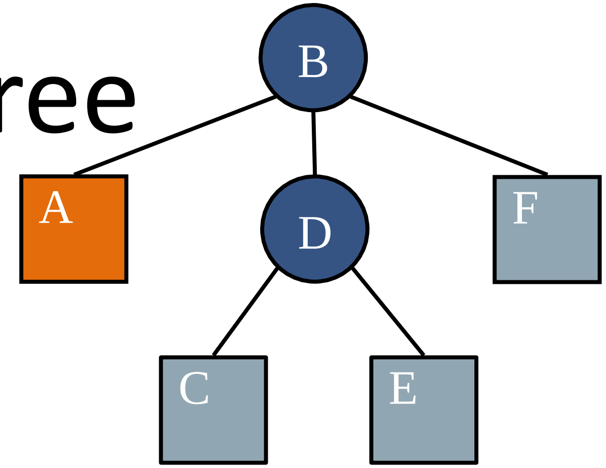



```
root = Tree("B")
```

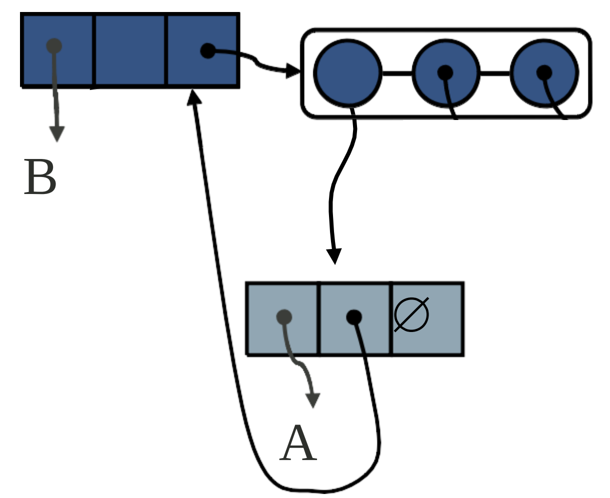


Tree.py

Structure for a Tree
- Add a child



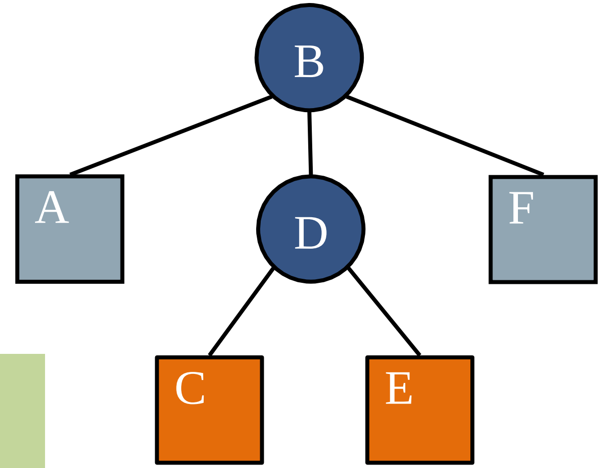

```python
class Tree:
    def __init__(self, data):
        self.children = []
        self.data = data
```



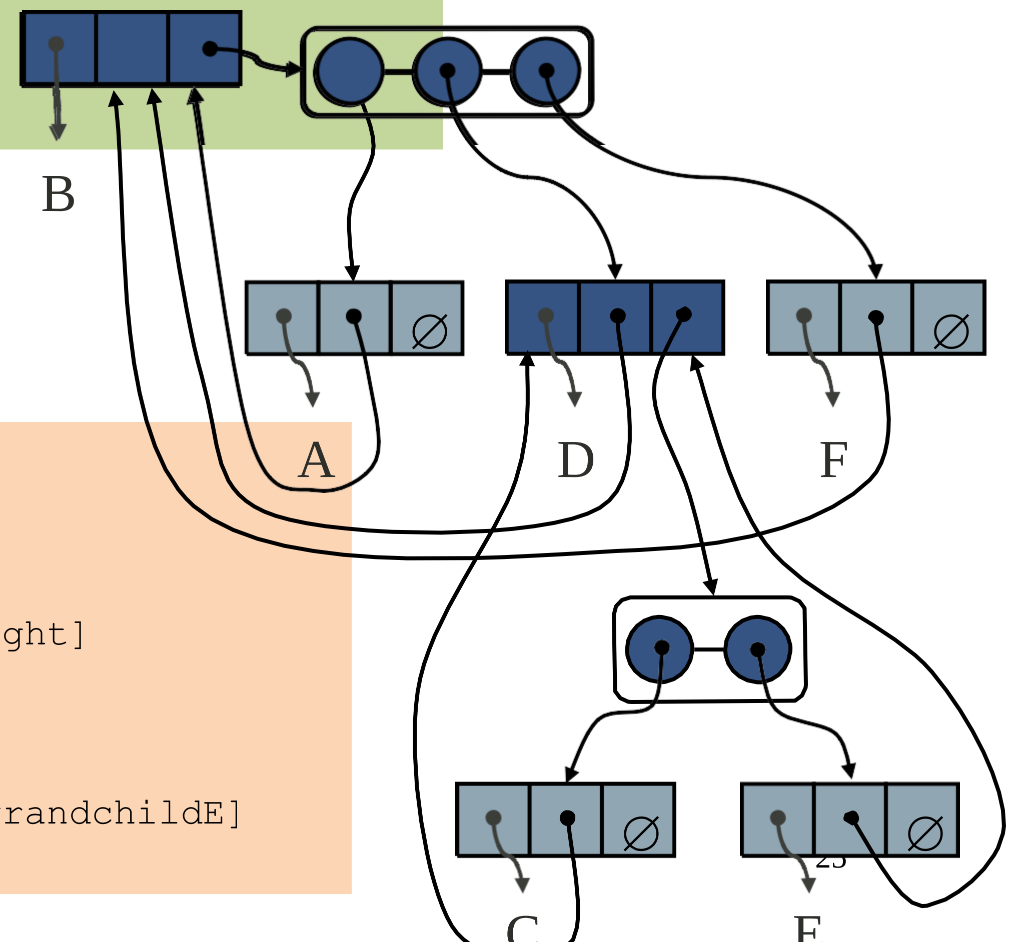

```python
left = Tree("A")
middle = Tree("D")
right = Tree("F")
root = Tree("B")
root.children = [left, middle, right]
```

Tree.py

Structure for a Tree - Add a grandchild




```python
class Tree:
    def __init__(self, data):
        self.children = []
        self.data = data
```



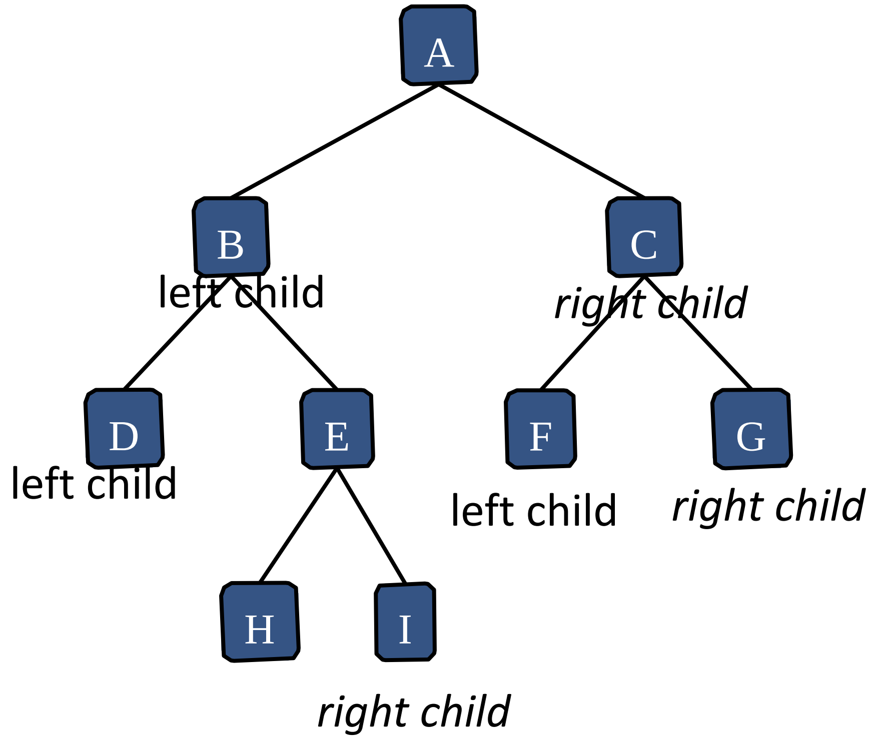

```python
left = Tree("A")
middle = Tree("D")
right = Tree("F")
root = Tree("B")
root.children = [left, middle, right]

grandchildC = Tree("C")
grandchildE = Tree("E")
middle.children = [grandchildC, grandchildE]
```

Binary Tree ADT and Implementation in Linked Nodes

Binary Tree

- A binary tree is a tree with the following properties:

1. Each *internal* node has *at most* **two** children
2. The children of a node are an ***ordered pair*** (*left* and *right*)
3. We call the children of an internal node *left child* and *right child*

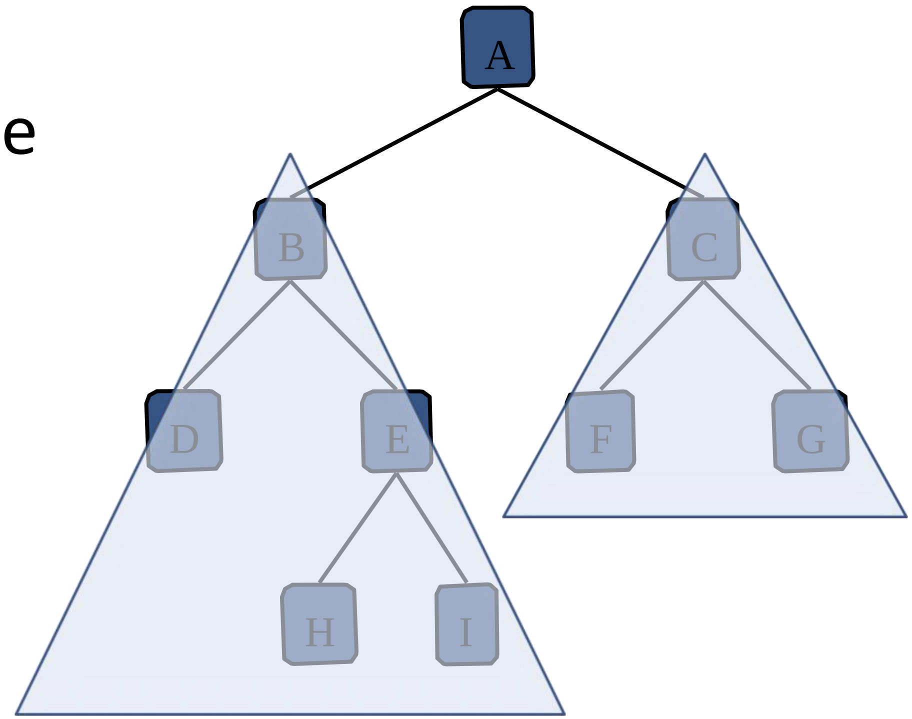


Binary Tree

Alternative *recursive definition*:

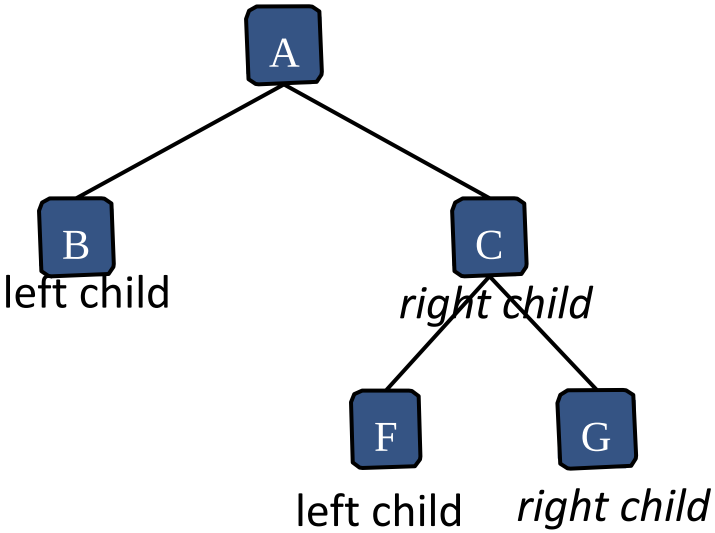
a binary tree is either

- a tree consisting of a single node, or
- a tree whose root has an ***ordered pair*** of *children*, each of which is a binary tree



BinaryTree Node

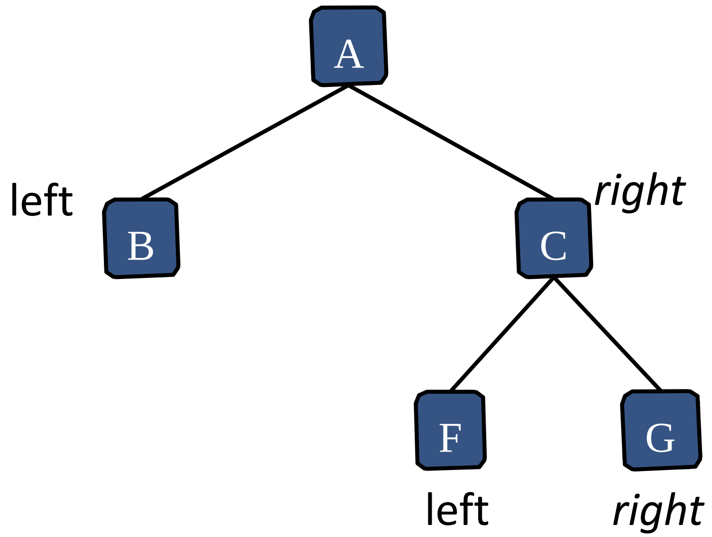
```python
class BinaryTreeNode:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data

a = BinaryTreeNode("A")
```



BinaryTree.py

BinaryTree Node

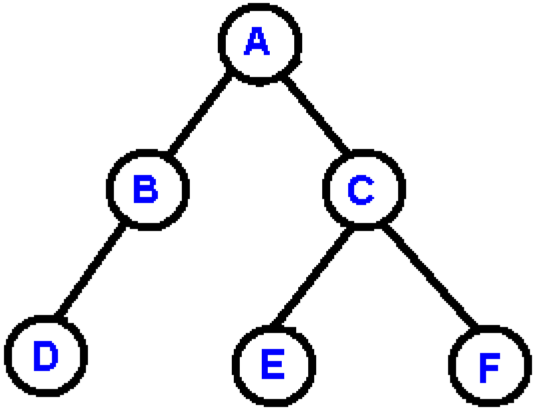
```
# Create Binary Tree
a = BinaryTreeNode("A")
b = BinaryTreeNode("B")
c = BinaryTreeNode("C")
a.setLeftChild(b)
a.setRightChild(c)

f = BinaryTreeNode("F")
g = BinaryTreeNode("G")
c.setLeftChild(f)
c.setRightChild(g)
```




```
    def setLeftChild(self,theLeftChild):
        left = theLeftChild
    def setRightChild(self,theRightChild):
        right = theRightChild
```

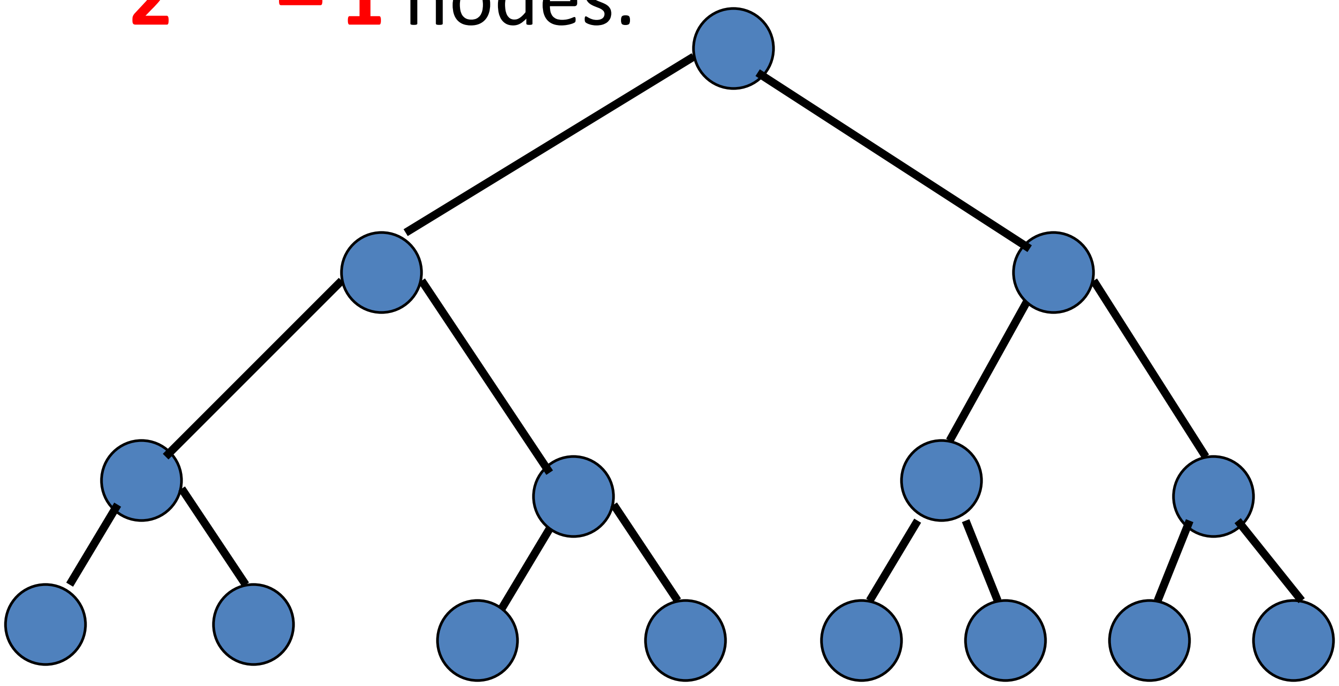

BinaryTree.py

Special
Binary Trees



Properties
- **Full Binary Tree**

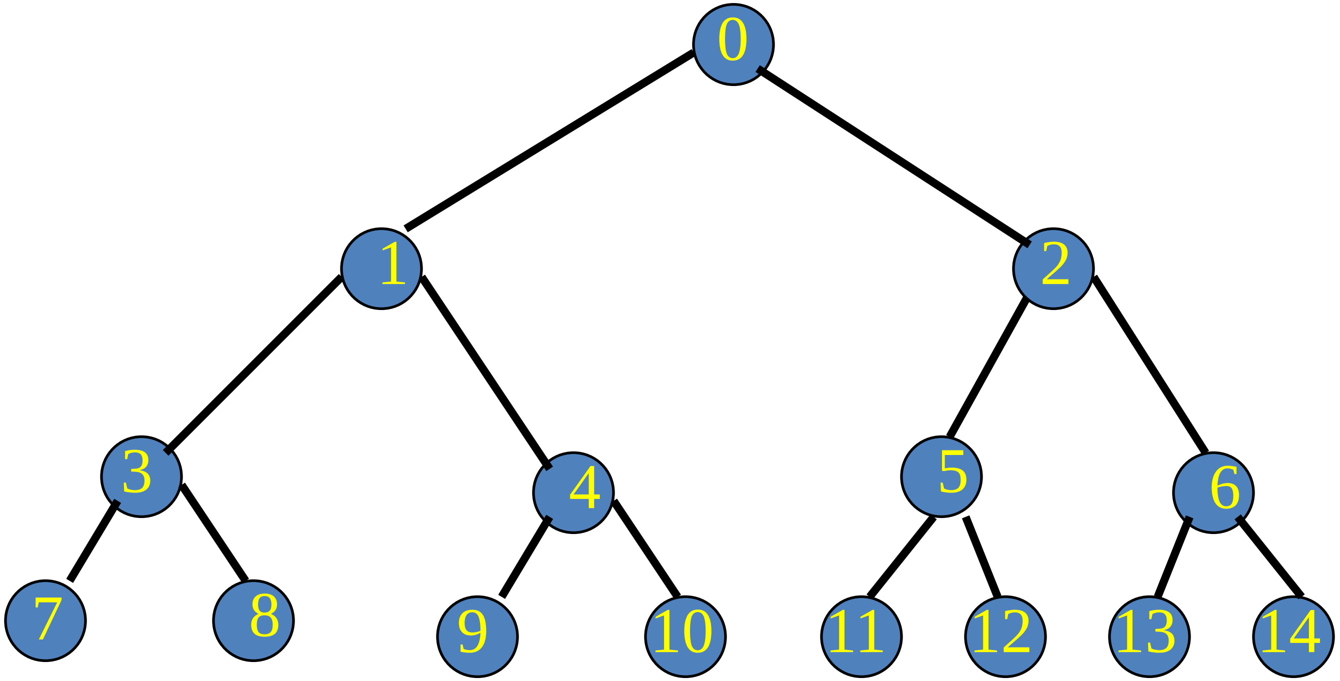
- A full binary tree of a given height **h** has **$2^{h+1} - 1$** nodes.



Height **3** full binary tree.

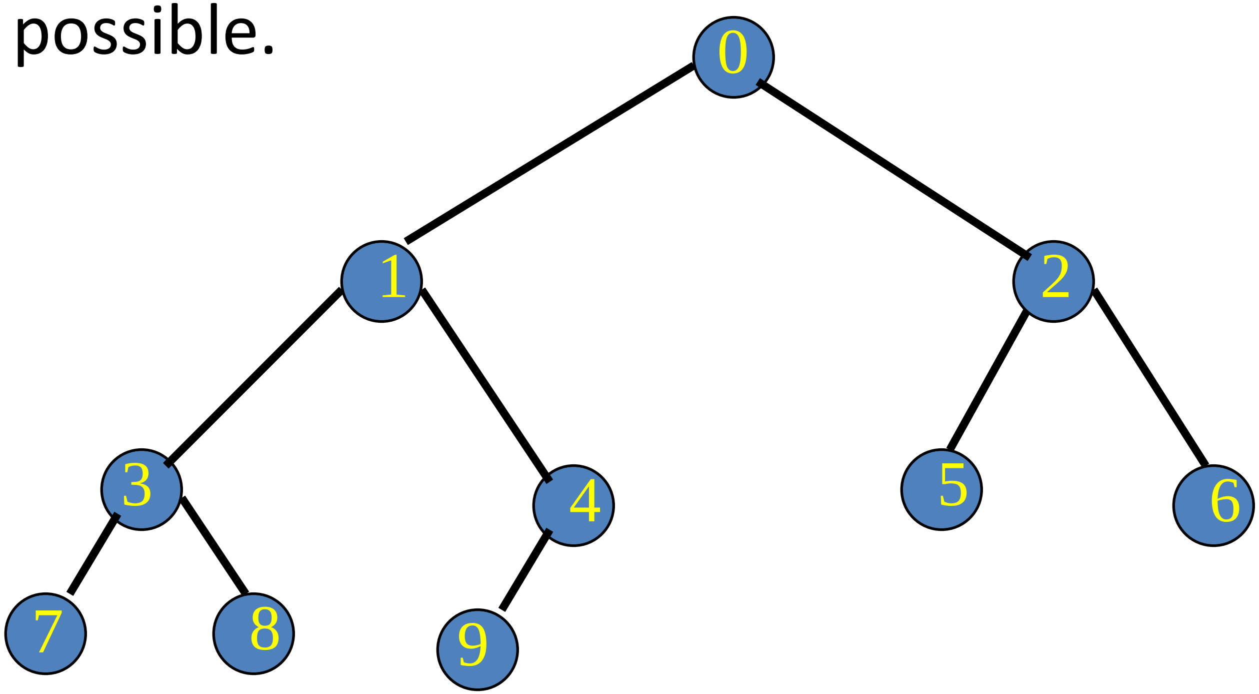
Numbering Nodes in a Full Binary Tree

- Number the nodes **0** through $\mathbf{2^{h+1} - 2}$.
(total $2^{h+1} - 1$ nodes)
- Number by levels from top to bottom.
- Within a level number from left to right.



Complete Binary Tree

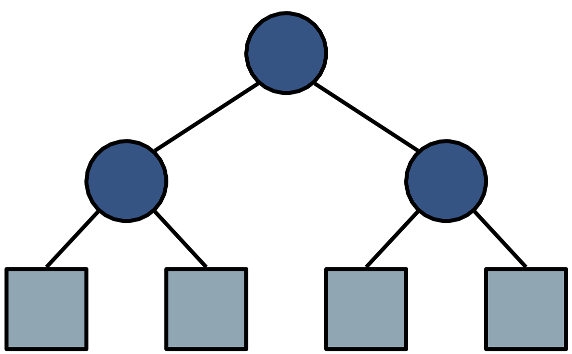
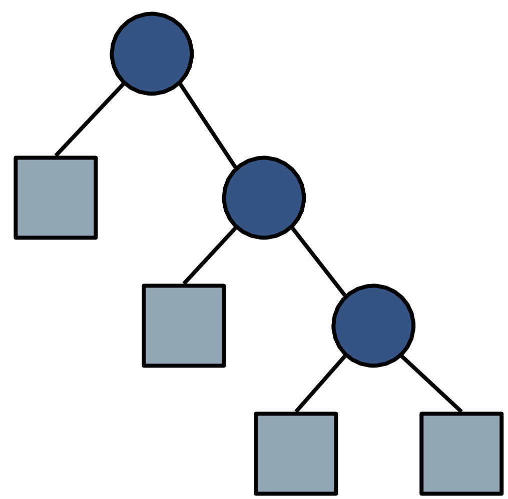
- all levels are completely filled except possibly the last level and the last level has all keys as **left** as possible.



- Example, Complete binary tree with 10 nodes.
- The size n of a complete binary tree of height h satisfies $\mathbf{2^h <= n <= 2^{h+1} - 1}$

Proper Binary Trees

Meaning: Each internal node has exactly 2 children

Proper Binary Trees

Given the following definitions:

n :number of total nodes

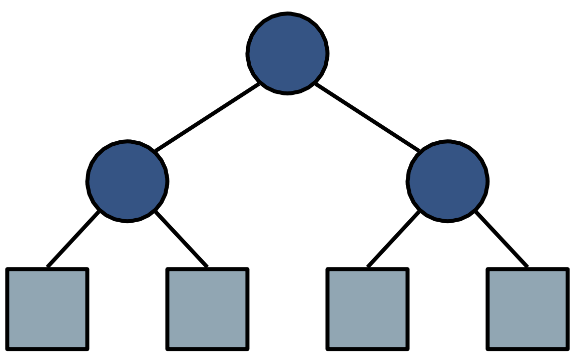
e :number of external nodes

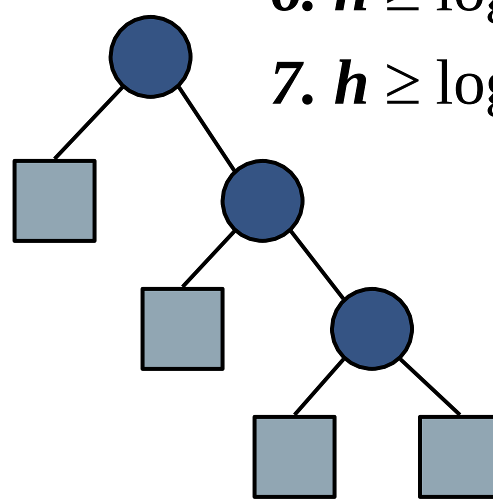
i :number of internal nodes

h :height (maximum depth of a node)

Properties:

1. $\boldsymbol{e} = \boldsymbol{i} + 1$
2. $\boldsymbol{n} = 2\boldsymbol{e} - 1$
3. $\boldsymbol{h} \leq \boldsymbol{i}$
4. $\boldsymbol{h} \leq (\boldsymbol{n} - 1)/2$
5. $\boldsymbol{e} \leq 2^{\boldsymbol{h}}$
6. $\boldsymbol{h} \geq \log_2 \boldsymbol{e}$
7. $\boldsymbol{h} \geq \log_2 (\boldsymbol{n} + 1) - 1$





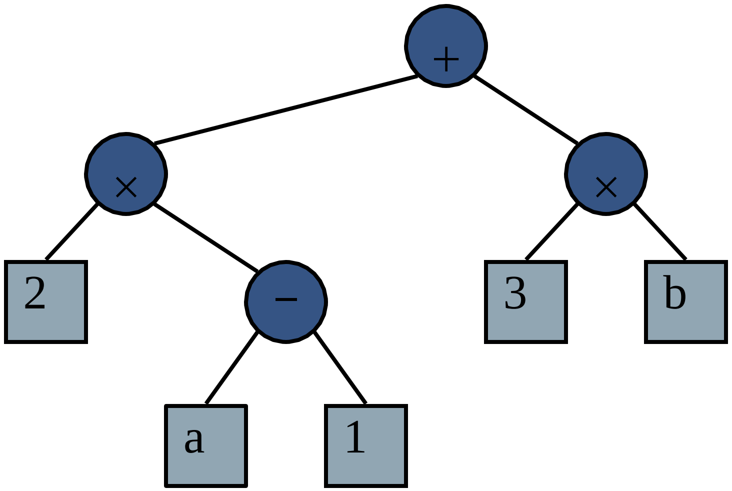
Examples of Usage of Binary Trees

Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: **operators**
  - external nodes: **operands**

- Example: arithmetic expression tree for the expression:

$(2 \times (a - 1) + (3 \times b))$

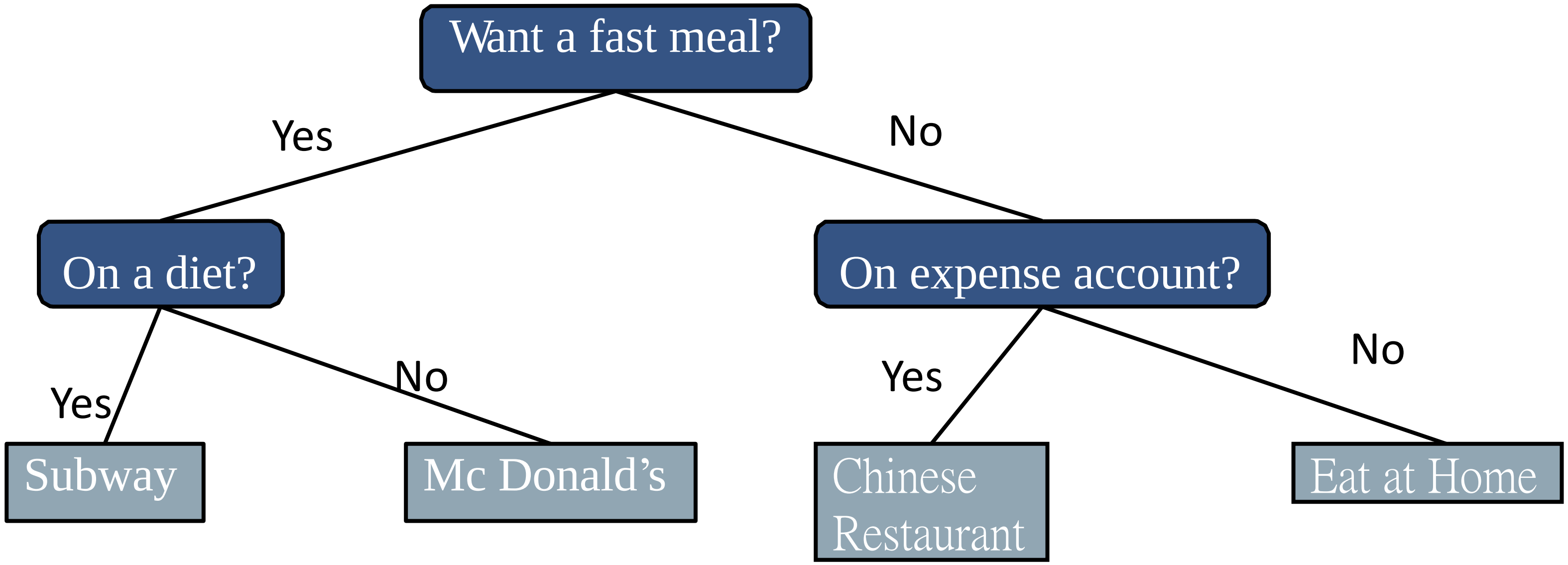


Decision Tree

Binary tree associated with a decision process

- internal nodes: questions with yes/no answer
- external nodes: decisions

- Example: dining decision



Traversals of a Tree

Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree.
- In a traversal, each element of the binary tree is **visited** exactly once.
- During the **visit** of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

Binary Tree Traversal Methods

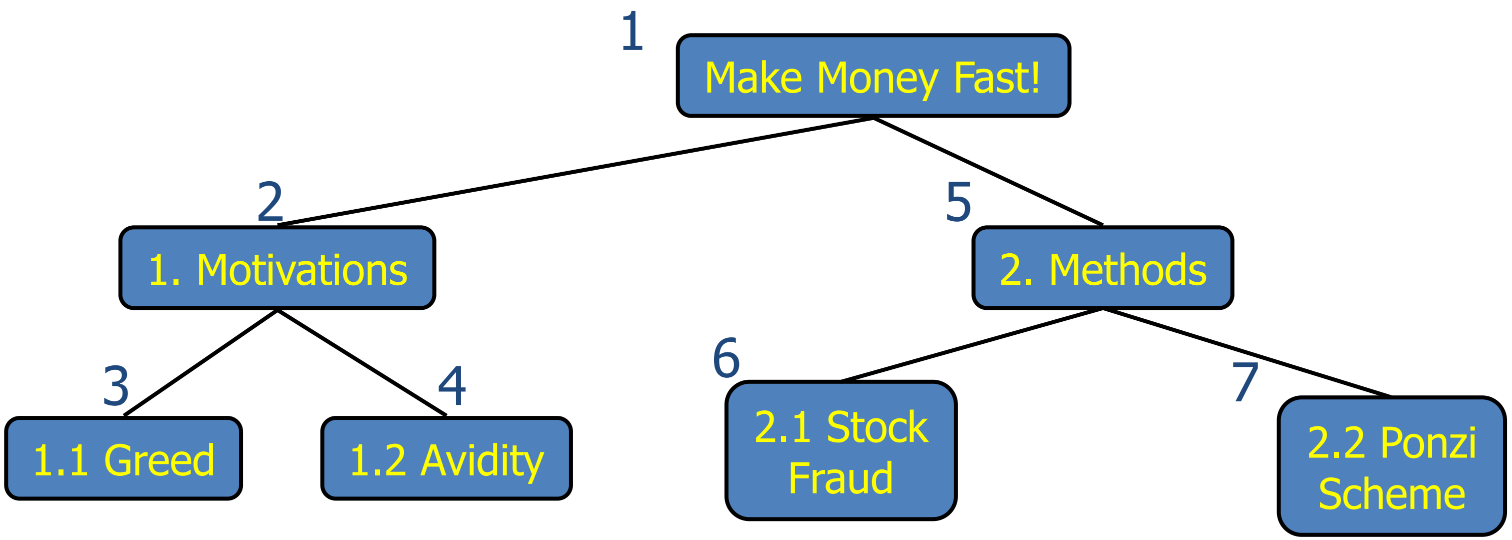
- Preorder (get prefix expression)
- Inorder (get infix expression)
- Postorder (get postfix expression)
- Level order

Preorder Traversals of a Binary Tree

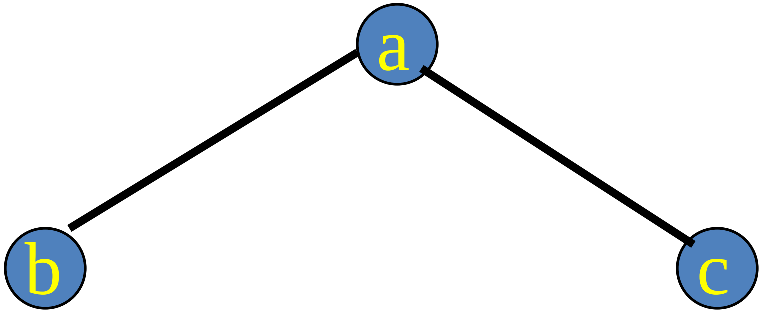
Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited **before its descendants**
- *Application*: print a structured document

Algorithm ***preOrder***(T,v)
  visit(v)
  for each child w of v
    preorder (T,w)

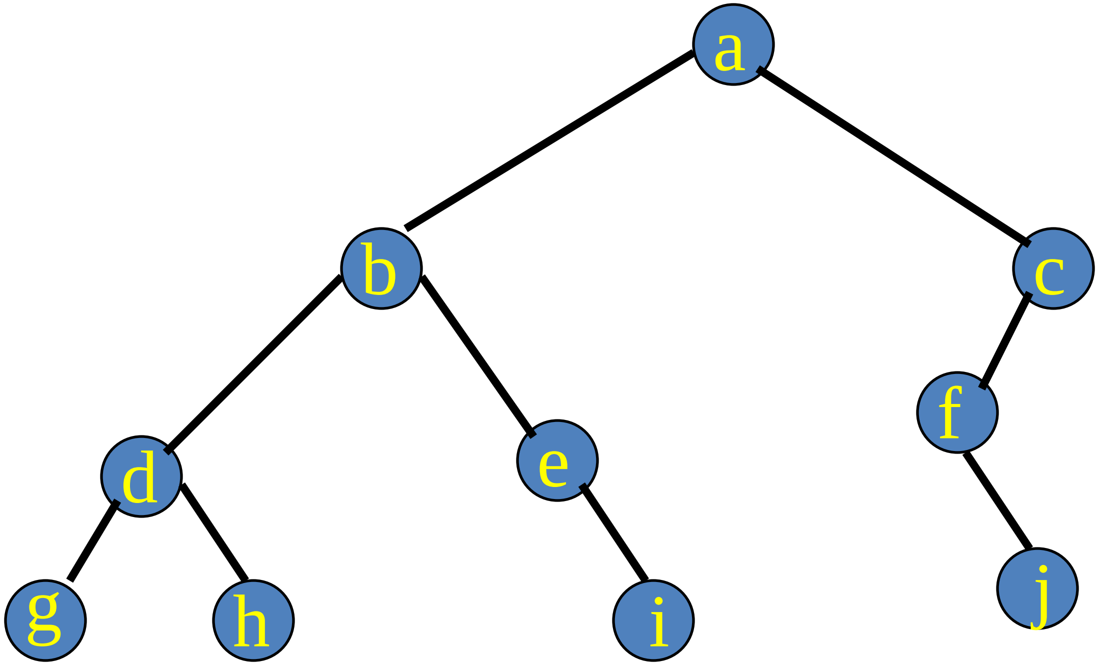


Preorder Example (visit = print)



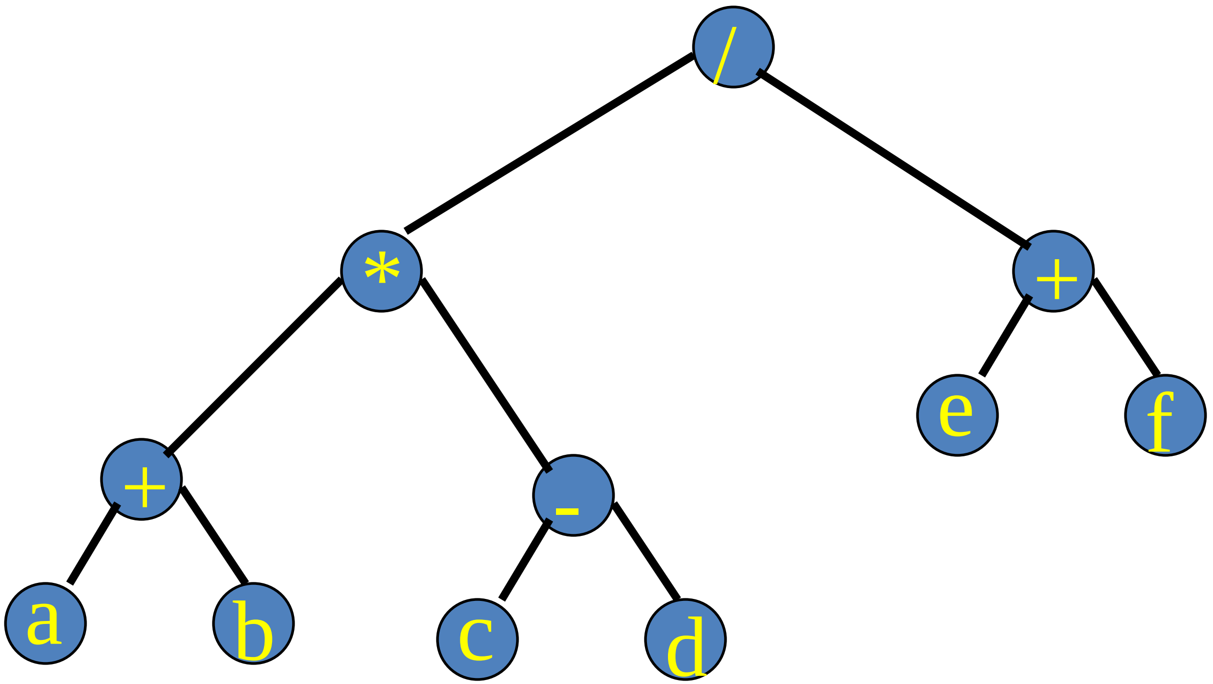
a b c

Preorder Example (visit = print)



a b d g h e i c f j

Preorder Of Expression Tree



/ * + a b - c d + e f

Gives **prefix** form of expression!

Preorder Traversal

```
def traversePreorder(self, root):
    if root is not None:
        print(root.data)
        self.traversePreorder(root.left)
        self.traversePreorder(root.right)
```

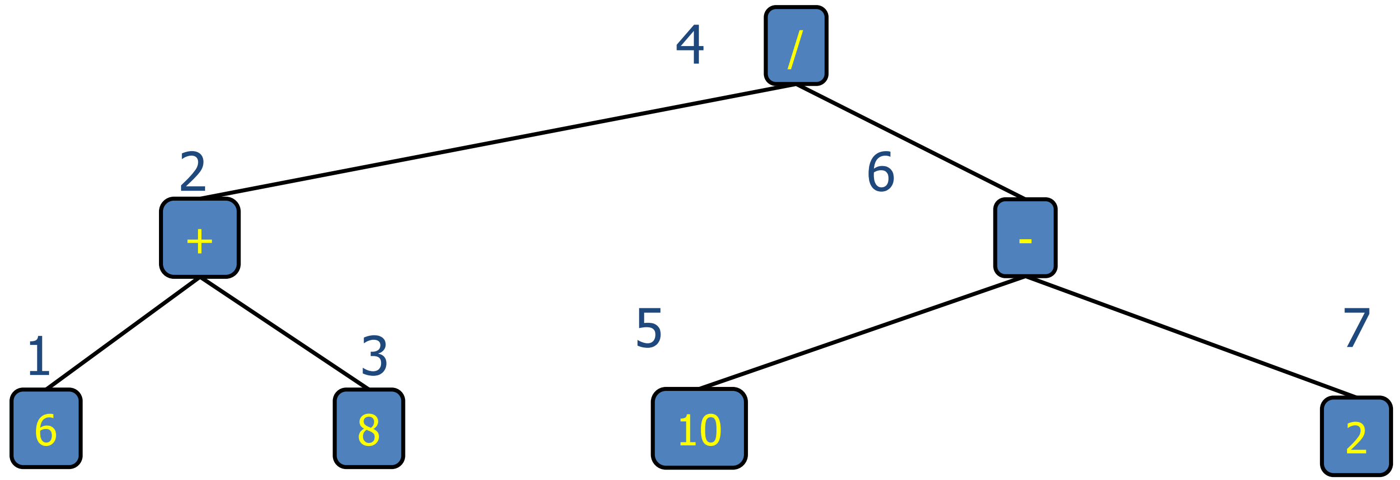
demo: BinaryTree2.py

Inorder
Traversals of a Binary Tree

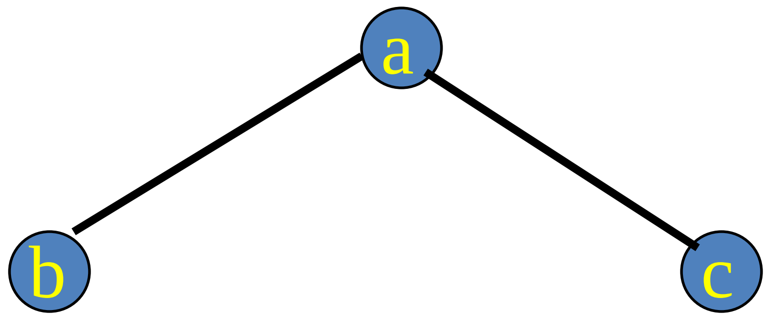
Inorder Traversal

- In an inorder traversal a node is visited **after its left** subtree and **before its right** subtree

Algorithm *inOrder*(*v*)
    if *isInternal* (*v*)
        inOrder (*leftChild* (*v*))
    visit(*v*)
    if *isInternal* (*v*)
        inOrder (*rightChild* (*v*))

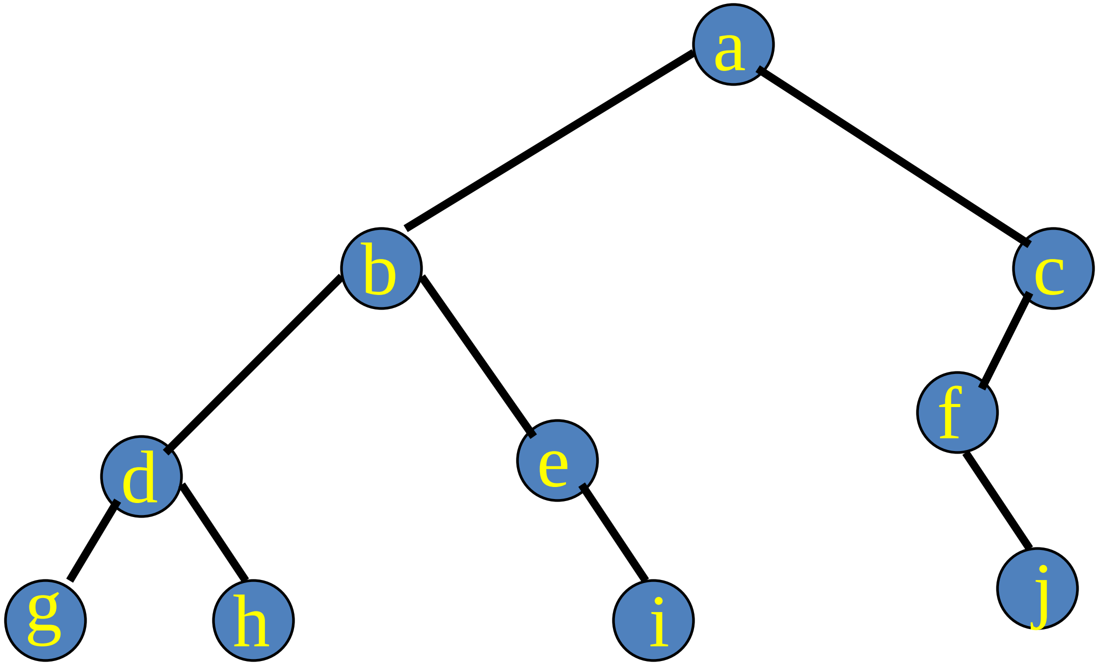


Inorder Example (visit = print)



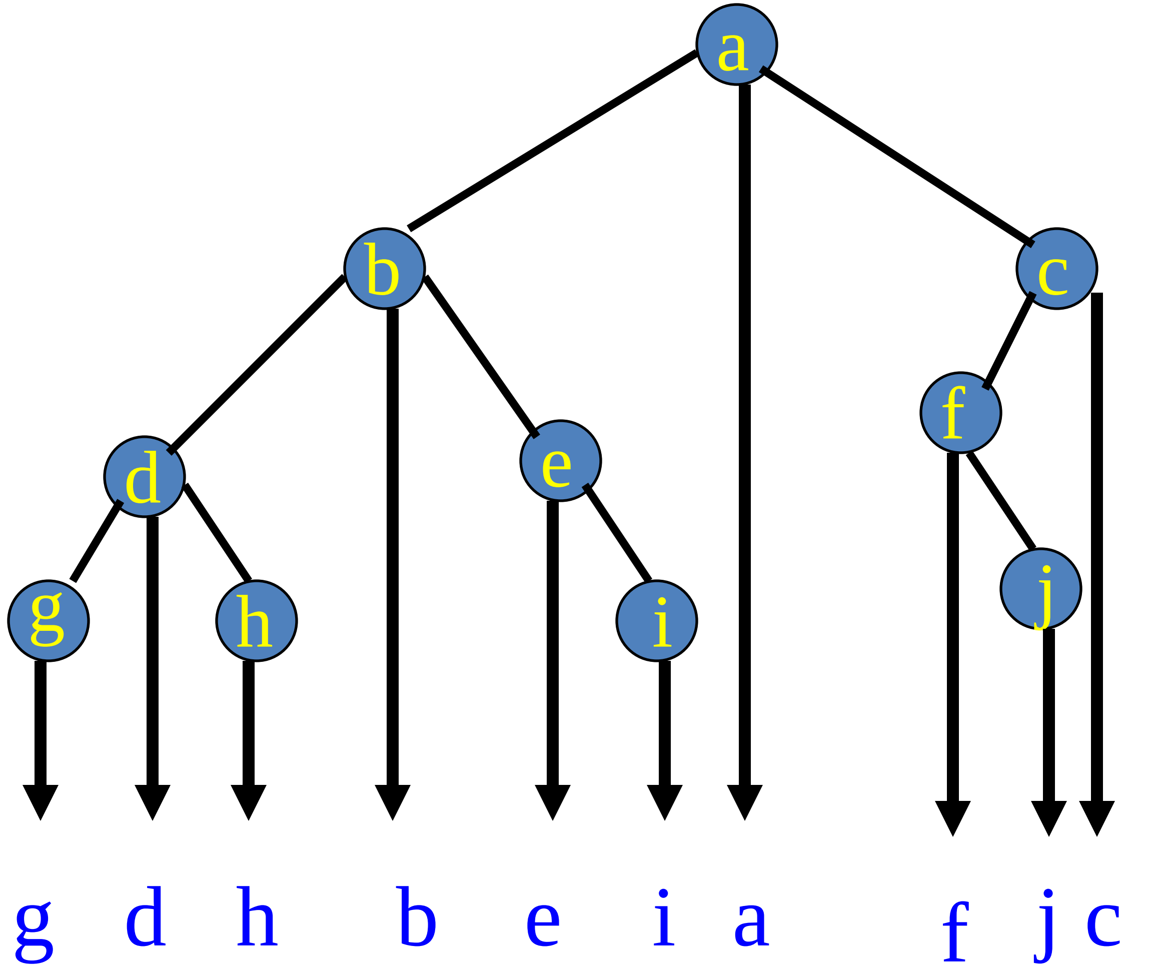
b a c

Inorder Example (visit = print)

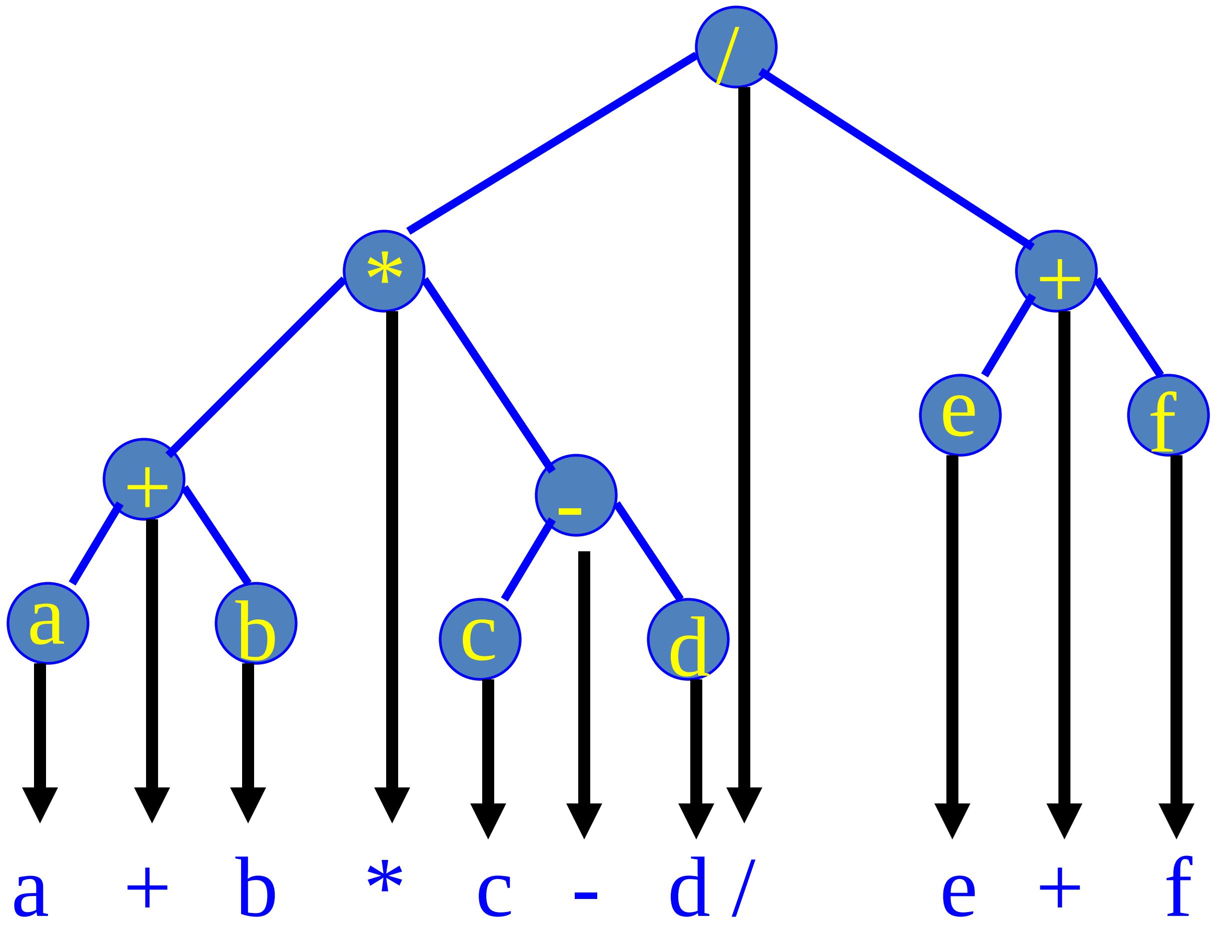


g d h b e i a f j c

Inorder By Projection



Inorder Of Expression Tree



Gives **infix** form of expression!

Inorder Traversal

```
def traverseInorder(self, root):
    if root is not None:
        self.traverseInorder(root.left)
        print(root.data)
        self.traverseInorder(root.right)
```

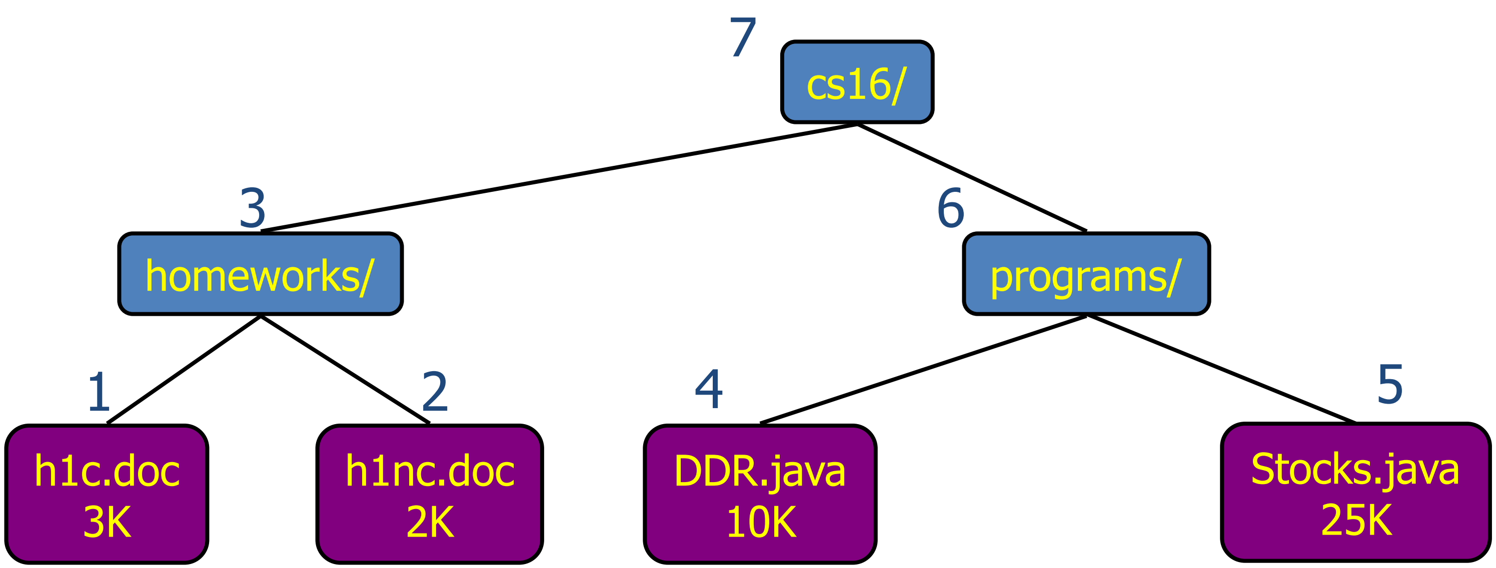
demo: BinaryTree2.py

Postorder Traversals of a Binary Tree

Postorder Traversal

- In a postorder traversal, a node is visited **after its descendants**
- Application: compute space used by files in a directory and its subdirectories

Algorithm *postOrder*(T,v)
    for each child w of v
        postOrder (T,w)
    visit(v)

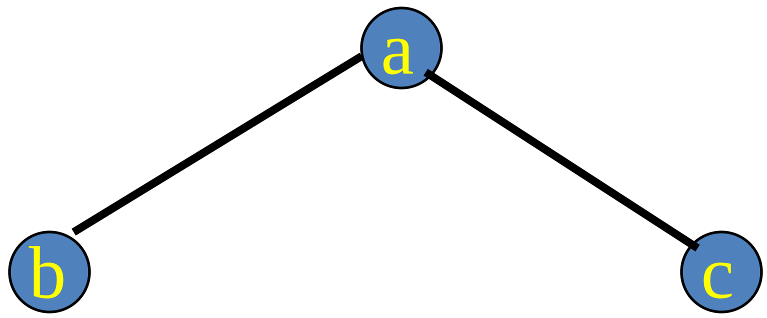


Postorder Traversal

```
def traversePostorder(self, root):
    if root is not None:
        self.traversePostorder(root.left)
        self.traversePostorder(root.right)
        print(root.data)
```

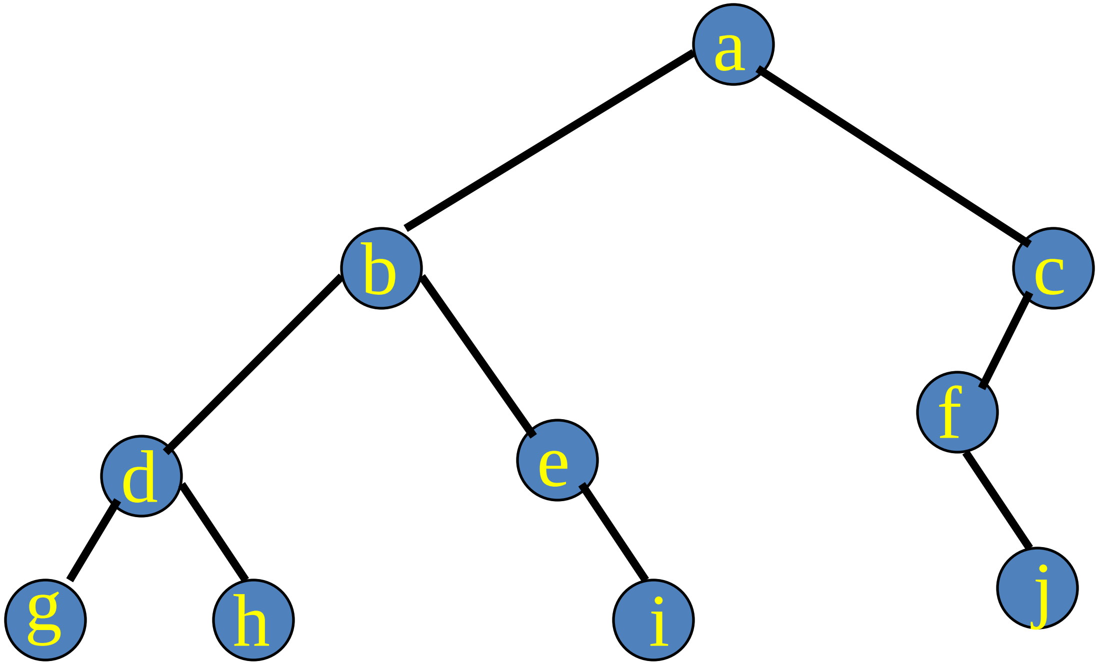
demo: BinaryTree2.py

Postorder Example (visit = print)



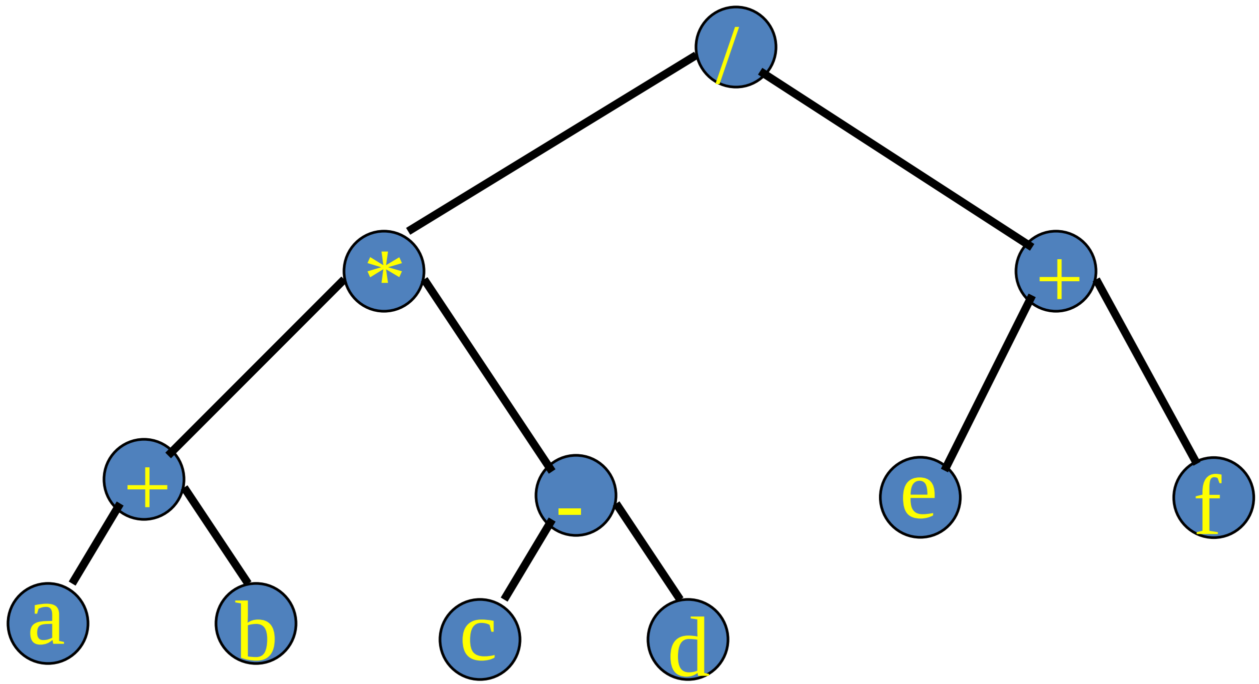
b c a

Postorder Example (visit = print)



g h d i e b j f c a

Postorder Of Expression Tree



a b + c d - * e f + /

Gives postfix form of expression!

Level order Traversals of a Binary Tree

Level Order

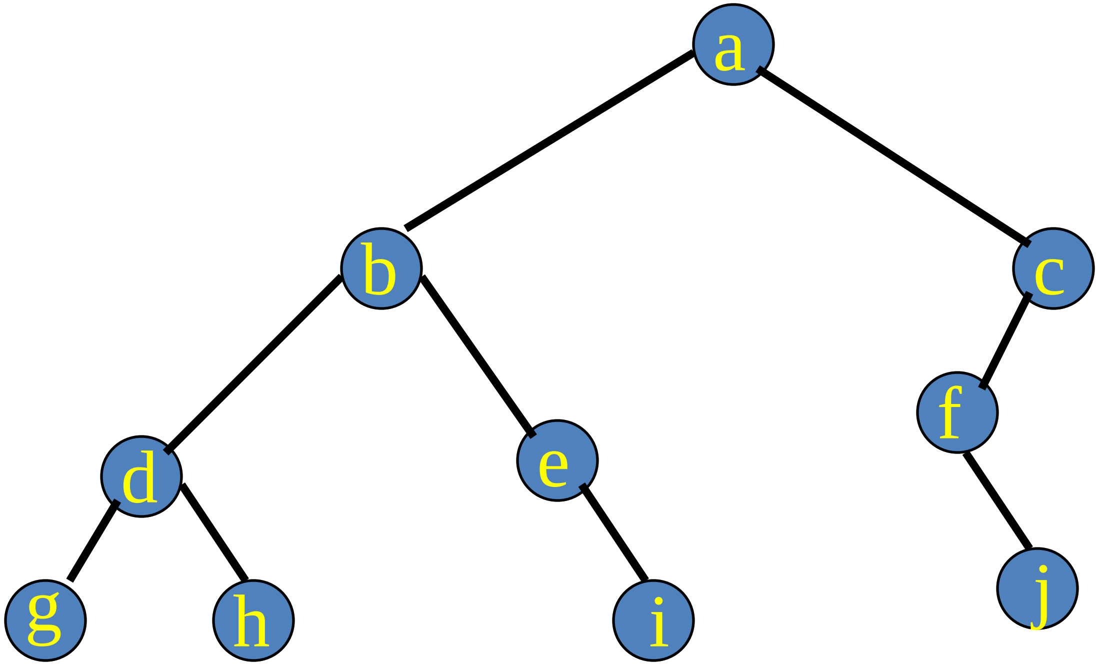
```
# Function to  print level order traversal of tree
def printLevelOrder(self, root):
    h = root.height(root)
    for i in range(1, h+1):
        self.printCurrentLevel(root, i)

# Print nodes at a current level
def printCurrentLevel(self, root , level):
    if root is None:
        return
    if level == 1:
        print(root.data,end=" ")
    elif level > 1 :
        self.printCurrentLevel(root.left , level-1)
        self.printCurrentLevel(root.right , level-1)

# Compute the height of a tree--the number of nodes
# along the longest path from the root node down to
# the farthest leaf node

def height(self, node):
    if node is None:
        return 0
    else :
        # Compute the height of each subtree
        lheight = self.height(node.left)
        rheight = self.height(node.right)

        #Use the larger one
        if lheight > rheight :
            return lheight+1
        else:
            return rheight+1
```

demo: BinaryTree2

Level Order

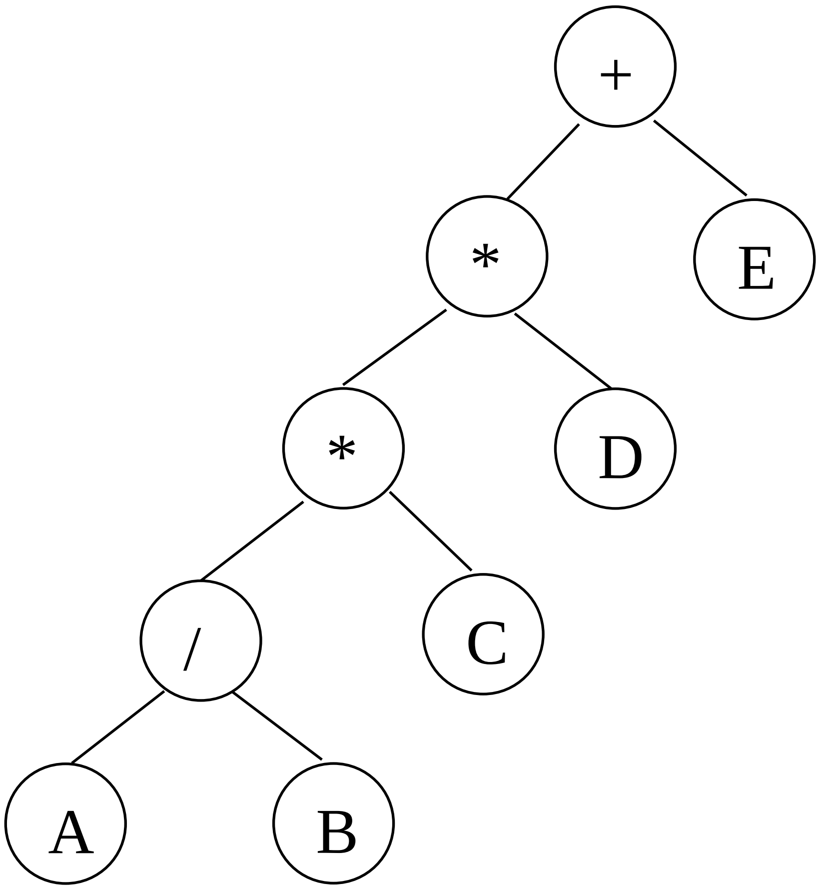
```
levelorder(root)
    q = empty queue
    t = root
    while not q.empty do
        visit(t)
        if t.left ≠ null
            q.enqueue(t.left)
        if t.right ≠ null
            q.enqueue(t.right)
         t := q.dequeue()
```

Level-Order Example (visit = print)



a b c d e f g h i j

Arithmetic Expression Using Binary Tree



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

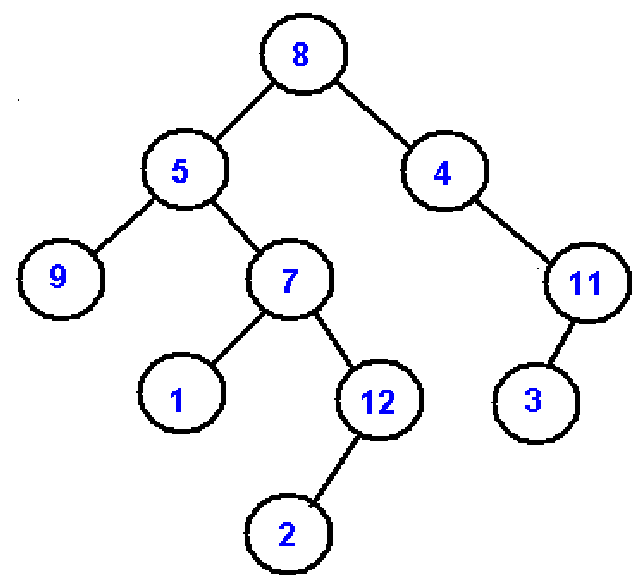
postfix expression

level order traversal

+ * E * D / C A B

Exercise

- As an example consider the following tree and its four traversals:
- PreOrder –
- InOrder –
- PostOrder –
- LevelOrder –



Construct the tree

- Preorder: JCBADEFIGH
- In-order: ABCEDFJGIH

Summary

- General Trees
  - Creation of Trees
- Binary trees
  - Binary Tree Implementation
  - Full, Complete and proper
  - Application: e.g. Decision Trees
- Tree Traversal
  - Pre-order,
  - postorder,
  - inorder or
  - level order